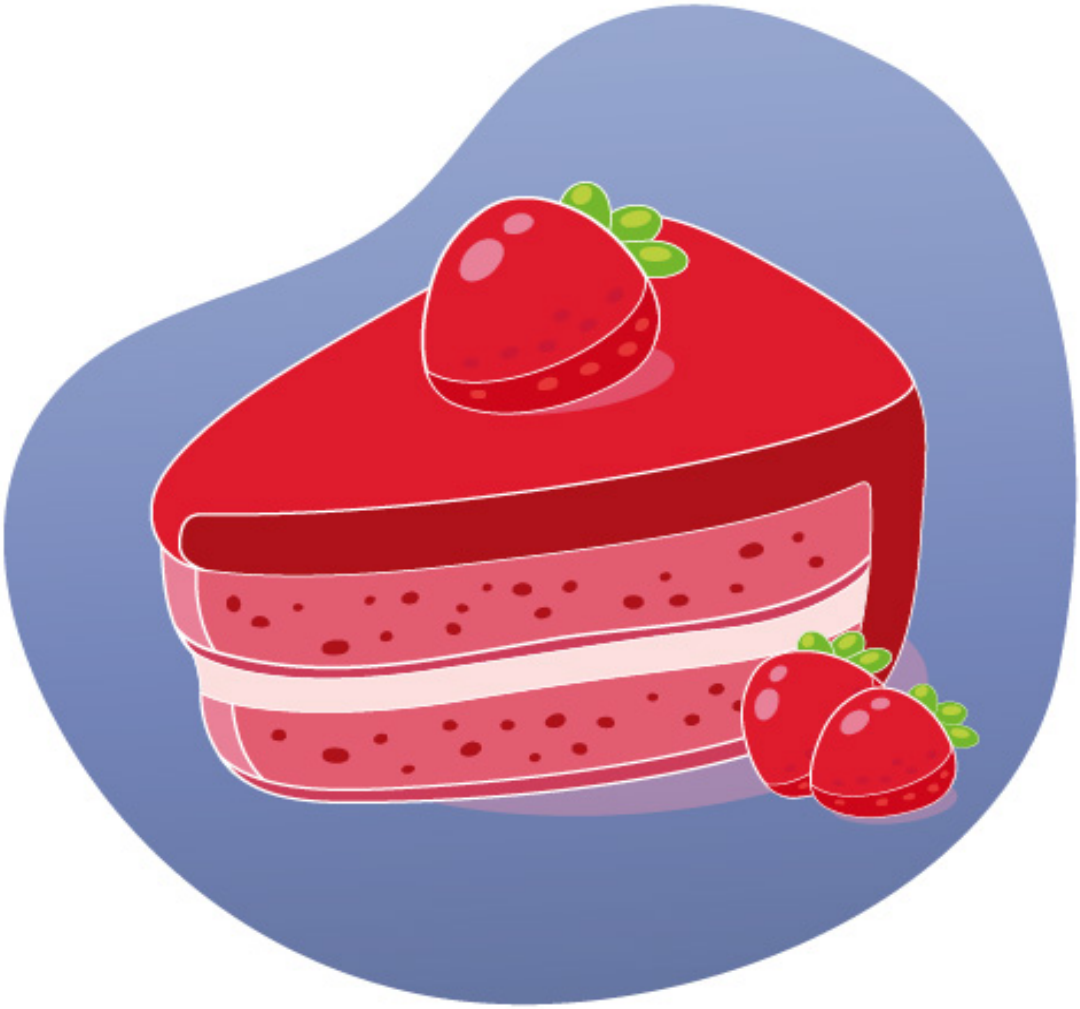
CakePHP

Construa aplicações web robustas rapidamente



Casa do Código | alura

JOSÉ CARLOS LIMA MOREIRA

Sumário

- ISBN
- Sobre o livro
- 1. O que é CakePHP
- 2. A aplicação AdPET
- 3. Migrations
- 4. Criando rotas
- 5. Middlewares
- 6. Controllers
- 7. Modelos
- 8. Queries
- 9. Helpers
- 10. Views
- 11. Forms
- 12. Autenticação
- 13. Finalizando o Projeto AdPET

ISBN

Impresso: 978-85-5519-332-3

Digital: 978-85-5519-333-0

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Primeiramente, agradeço, com enorme amor, ao meu Pai, que está nos céus e que está comigo em todas as noites de trabalho, em cada pensamento, em cada palavra digitada — muito obrigado, meu bom Pai, meu Criador.

Agradeço à minha esposa, Deise Cristina, que torceu e contribuiu com a realização desta obra, e aos meus filhos, Mariah Christina e Johan Carlos. Agradeço à minha mãe, Sizesnanda Lima, minha primeira professora, presente em toda a minha formação intelectual.

Em memória de Vovô Alcides e Vovô Bambo, Vovó Maria, Quintino Carneiro e José Lima de Sousa — que Deus os tenha em seus braços. Saudades.

Sobre o livro

Acreditamos que a melhor maneira de consolidar o conhecimento é por meio da prática. A cada capítulo, teremos conceitos do framework CakePHP e códigos para exemplificar e reforçar o aprendizado. Para praticar os conhecimentos descritos, uma aplicação será construída incorporando a teoria ensinada nos capítulos — basicamente, é um *hands-on* que tem o objetivo de deixar a leitura mais atrativa e fácil.

A aplicação que será desenvolvida aqui é a AdPET, um sistema para adoção de pets. O usuário do sistema poderá cadastrar um animal para adoção ou pedir para adotar um animal que foi anunciado. A aplicação vai permitir uma interação social entre os usuários para trocar informações sobre o animal. Para construir o site, vamos precisar modelar tabelas no banco de dados, aprender como autenticar usuários, fazer uploads de imagem, entre outras coisas. Tudo isso será abordado gradualmente no livro e, ao final, além de termos explorado as funcionalidades do CakePHP, teremos uma aplicação capaz de:

- Fazer autenticação;
- Criar postagens de animais;
- Criar formulários de contato com upload de imagens;
- Fornecer a funcionalidade Chat;
- Fornecer a funcionalidade Comentários.

Este livro tem também um cunho social: despertar nos leitores e leitoras o entendimento de que é possível utilizar o seu potencial, as suas habilidades, o seu conhecimento, a sua criatividade, além da participação da população, para gerar ferramentas que ajudem a melhorar a vida nas cidades.

Para quem é este livro

Este livro é destinado a profissionais que desenvolvem projetos utilizando PHP e que já possuem conhecimentos básicos de

Programação Orientada a Objetos. Podemos destacar que o conteúdo deste livro é para:

- Leitores e leitoras que pretendem aprender a tecnologia e aplicá-la nos seus futuros projetos ou na empresa em que trabalham;
- Profissionais que conhecem o CakePHP, mas pretendem melhorar ou revisar seus conhecimentos e conceitos sobre a tecnologia;
- Profissionais que já utilizam alguma versão anterior do framework e que precisam migrar para a nova versão 4 do CakePHP.

Como a ênfase do livro é no framework CakePHP, seria fundamental que os leitores e as leitoras que não possuem prática com o uso de classes, objetos e teorias relacionadas fizessem leituras complementares, a fim de ter um melhor aproveitamento do conteúdo aqui abordado.

Quem é o autor?

Oi, pessoal! Meu nome é José Carlos, nasci em uma bela cidade do Ceará chamada Paracuru. A minha idade? É melhor deixá-la escondida, mas posso dizer que não sou tão novo a ponto de não poder ensinar algo, nem tão velho a ponto de não ter mais nada a aprender. Hoje, eu tenho uma bela família para cuidar e proteger: sou pai de duas crianças inteligentes e esposo de uma linda mulher. Desde pequeno, pelos incentivos da minha mãe e da minha linda vovó Hermelinda, aprendi a gostar de estudar e entender o seu valor. Tive o meu primeiro contato com programação aos 12 anos de idade, com a linguagem Logo (aquela da tartaruguinha). Foi aí que despertou o meu interesse e a aptidão pela área de tecnologia da informação.

A minha formação tecnológica acadêmica começou com a minha graduação em mecatrônica, concluída em 2008 no Instituto Federal do Ceará (IFCE). Meu trabalho de conclusão de curso foi sobre as

áreas de processamento digital de imagens e redes neurais artificiais. Em 2010, fiz uma especialização em tecnologia da informação pela Universidade Federal do Ceará (UFC), onde desenvolvi uma ferramenta web para traçar a rota dos itinerários dos ônibus da cidade de Fortaleza/CE, uma ferramenta de cunho social. Enquanto escrevo esse livro, estou fazendo o mestrado acadêmico em ciências da computação, também pelo IFCE.

No campo profissional, tenho 18 anos de experiência e sempre dediquei o meu esforço e conhecimento na área de desenvolvimento de software usando as mais variadas tecnologias para as plataformas web, desktop e móvel. Meu objetivo profissional é aprimorar sempre a qualidade do meu trabalho por meio de estudos, técnicas e tecnologias para criar softwares melhores, no menor tempo possível e que surpreendam as expectativas dos usuários. Hoje, após trabalhar em empresas no Brasil, desenvolvo software para empresas estrangeiras, principalmente dos Estados Unidos e de países da Europa.

Capítulo 1
O que é CakePHP

CakePHP é um framework usado no desenvolvimento de aplicações para a plataforma web escrito em PHP. Possui código-fonte aberto e uma comunidade grande e participativa que influencia muito na divulgação e no aperfeiçoamento do projeto.

1.1 Por que usar CakePHP em meus projetos?

Quando precisamos desenvolver um sistema, principalmente para a plataforma web, por mais básico que seja, sempre encontramos tarefas comuns que precisam ser desenvolvidas, tais como autenticar usuários, validar formulários, salvar informações em bancos de dados, entre outras que sempre fazem parte do escopo de muitos projetos. A cada novo sistema, refazer as mesmas tarefas que poderiam ser facilmente reutilizadas é um erro que, além dos custos, impacta no tempo de desenvolvimento desse produto.

A engenharia de software propõe a reutilização de código como solução para esse problema a fim de melhorar o processo de desenvolvimento, alcançar os objetivos de forma mais rápida e ainda garantir a qualidade do produto. Nesse sentido, os desenvolvedores de software podem usar os frameworks, que nada mais são do que ferramentas com funcionalidades genéricas que viabilizam o ganho de produtividade pelo simples fato de permitirem que os desenvolvedores direcionem os esforços para tarefas que têm um maior valor agregado ao negócio, além de melhorar a organização e a manutenção do projeto.

Como o PHP está entre as dez linguagens mais utilizadas no mundo — de acordo com o site TIOBE Index, em 2022, ela é a décima

colocada no ranking das linguagens —, certamente podemos citar o CakePHP como uma ferramenta ideal para os desenvolvedores PHP construírem sistemas para a plataforma web. Ele possui código aberto, licença livre e permite desenvolver um código flexível, estruturado, além de oferecer recursos como:

- Arquitetura MVC;
- Componentes para trabalhar com e-mail;
- Cookie;
- Segurança;
- Validação de dados;
- Integração com diferentes bancos de dados;
- *Scaffolding* para a criação rápida de páginas.

Nos capítulos seguintes, sabendo que aqui usaremos CakePHP para desenvolver uma aplicação web, a AdPET, este livro apresentará recursos do framework que vai permitir desenvolver toda a aplicação, desde a sua instalação até a criação da aplicação completa, com acesso a dados, realização de consultas e outras interações com o usuário.

Capítulo 2
A aplicação AdPET

A cada capítulo, novos conceitos e conhecimentos sobre o CakePHP serão apresentados. Para reforçar o aprendizado e aplicar o conteúdo, aos poucos desenvolveremos uma aplicação web, que carinhosamente chamamos de AdPET. Essa aplicação tem um cunho social: usar o nosso conhecimento para gerar soluções que possam ajudar de alguma forma o local em que vivemos e a sociedade. Nesse caso, a aplicação AdPET é um sistema que permite anunciar animais para adoção. O primeiro passo para podermos criar essa aplicação é instalar o CakePHP, cujos passos serão descritos a seguir.

2.1 Instalando o CakePHP

A instalação do CakePHP não requer conhecimentos avançados. Neste primeiro momento, não é preciso configurar um servidor web para iniciar o desenvolvimento. Trataremos sobre o banco de dados mais adiante. Caso já queira trabalhar com outro servidor, o framework funciona com os principais servidores web, como XAMPP, WAMP, Nginx, LightHTTPD e Microsoft IIS. Porém, antes de prosseguir com a instalação, precisamos garantir que nosso ambiente de desenvolvimento tenha os seguintes requisitos instalados:

- PHP 7.2 ou superior;
- MySQL 5.6 ou superior;
- A extensão `mbstring`, essencial para conversão e codificação de caracteres;
- A extensão `intl`, que fornece módulos para trabalhar com internacionalização;

- A biblioteca PDO PHP, que fornece uma camada de abstração em relação à conexão com o banco de dados;
- A extensão `openssl` ;
- A extensão `simplexml` , que fornece métodos para converter XML em objetos que permitem que suas propriedades sejam acessadas mais facilmente;
- Composer;
- Git, que é um requisito para o Composer funcionar corretamente.

> **O QUE É O COMPOSER?**
>
> O Composer é um gerenciador de dependência que traz para o mundo PHP toda a praticidade encontrada no Ruby com o Gem; no Node.js, com o uso do NPM; ou no Python, com o PIM. Com ele, é possível declarar, reutilizar e atualizar com o mínimo de esforço as dependências de um projeto PHP. O CakePHP recomenda o uso do Composer como ferramenta oficial para a instalação do framework. Todos os pacotes baixados do Composer vêm do repositório https://packagist.org/.

Para quem trabalha com Linux ou macOS, para instalar o Composer, basta executar o comando usando o `curl` :

```
curl -s https://getcomposer.org/installer | php
```

Em seguida, mova o arquivo `composer.phar` para o diretório `bin` do sistema de arquivos:

```
mv composer.phar /usr/local/bin/composer
```

Caso não tenha o aplicativo `curl` , uma alternativa para instalar é usar o comando `php` . Essa é uma opção bem legal, pois também funciona no Windows:

```
php -r "readfile('https://getcomposer.org/installer');" | php
```

Para quem está trabalhando no Windows e pretende usar o Composer, podemos baixar o instalador (disponível em https://github.com/composer/windows-setup/releases/) e instalar a partir do arquivo `.exe`. Esta é a opção mais recomendada para Windows.

É possível também fazer o download direto da última versão do `composer.phar` em https://getcomposer.org/composer.phar.

2.2 Criando um novo projeto

Para criar uma nova aplicação CakePHP, supondo que o Composer esteja devidamente instalado, devemos executar o seguinte comando:

```
composer create-project --prefer-dist cakephp/app:~4.0 [nome_do_projeto]
```

O comando `create-project` faz o Composer criar um projeto a partir do pacote `cakephp/app` versão 4, que é um esqueleto para aplicações usando CakePHP. Ele recebe como parâmetro o nome do projeto, que também definirá o nome do diretório que armazenará todos os arquivos da aplicação.

O argumento `--prefer-dist` informa que as dependências devem ser baixadas do repositório — pode ser o GitHub ou outro — do pacote de distribuição do CakePHP, caso esteja disponível. As dependências geralmente vêm compactadas em um arquivo .zip, o que deixa o download mais rápido, além de não ser necessário baixar cada arquivo do código-fonte do projeto individualmente.

Em outras palavras, o `create-project` seria a junção de dois comandos consecutivos, um `git clone` seguido de um `composer install`, porém tudo gerenciado pelo Composer. Isso evita possíveis erros com o Git ou outro problema que possa existir no ambiente de desenvolvimento durante a criação do projeto.

Especificando a versão do CakePHP

Este livro vai trabalhar com a última versão do framework, a versão 4.X; porém, é possível informar para o Composer qual versão do CakePHP pretendemos obter apenas adicionando o número da versão desejada na linha de comando. Por exemplo, se eu quero baixar a versão 3.8, então o comando seria:

```
php composer.phar create-project --prefer-dist cakephp/app:3.8
[nome_do_projeto]
```

Outra maneira de instalar é usando o *Oven*, que é basicamente um script de instalação feito em PHP. Este script vai verificar as dependências necessárias e criará uma aplicação CakePHP. As instruções de uso e o código-fonte podem ser acessados em https://github.com/CakeDC/oven.

Como podemos instalar sem o Composer?

Caso exista algum motivo que impossibilite a instalação com o Composer, é possível fazer o download da última versão do CakePHP no GitHub. O link para download é https://github.com/cakephp/cakephp/tags. Vale lembrar que, dessa maneira, não é possível obter atualizações dos pacotes automaticamente — função provida pelo Composer.

Embora existam outros meios para criar um projeto CakePHP, para o projeto deste livro, vamos utilizar o Composer, que é a opção mais recomendada para este fim. Como foi demonstrado, ele é flexível ao permitir a utilização da versão desejada do framework.

Entendendo a estrutura do projeto

Após o projeto CakePHP ser criado e antes de iniciar o desenvolvimento do código, vamos conhecer a estrutura de arquivos e diretórios. Saber onde podemos customizar, criar novos arquivos ou mudar alguma configuração na árvore de diretórios é fundamental. O CakePHP segue o padrão PSR-4 para carregamento e saber como caminhar dentro da aplicação é importante para garantir qualidade no código que vai ser desenvolvido.

> **Um pouco sobre o padrão PSR-4**
>
> O PSR-4 define regras para *autoload* de classes, ou seja, é um padrão para carregar automaticamente classes PHP sem a necessidade de usar declarações como `require` e `include`. O PSR-4 é uma melhoria do padrão PSR-0, que também define regras de *autoloading*, mas que atualmente está em desuso.

Podemos dividir o detalhamento das estruturas em duas partes. A primeira, que compõe a raiz da aplicação, tem a seguinte estrutura de diretórios e arquivos:

- `bin` : contém arquivos internos do framework;
- `config` : é onde vamos adicionar configurações para o projeto, como conexão com o banco de dados, configuração do servidor SMTP;
- `logs` : por padrão, é usado para guardar os logs da aplicação, caso não seja especificado um novo local;
- `plugins` : local onde serão instalados os plugins;
- `src` : é aqui que vamos organizar todo o código relacionado à lógica da aplicação;
- `template` : contém as páginas, layouts e elementos que podem ser utilizados para criar uma página;
- `tests` : diretório onde as classes de testes devem ser criadas;
- `tmp` : usado para guardar arquivos temporários, como arquivos de sessões, cache, arquivo de modelos etc.;

- `vendor` : todas as dependências do projeto são mantidas aqui. O Composer salva as dependências neste diretório;
- `webroot` : diretório público, contém os arquivos que podem ser acessados pelo usuário externo;
- `.htaccess` : arquivo de configuração do Apache;
- `composer.json` : arquivo de configuração do Composer;
- `index.php` : arquivo padrão para inicializar a aplicação;
- `README.md` : pode conter informações adicionais sobre o projeto.

A segunda parte refere-se ao diretório `src` descrito anteriormente, porém merece uma atenção especial, porque é onde vamos realmente trabalhar escrevendo os arquivos da nossa aplicação. Esse diretório é composto por:

- `Command` : contém os comandos do console do seu aplicativo;
- `Console` : contém as classes que executam comandos e tarefas;
- `Controller` : contém as classes *controllers*;
- `Form` : armazena as classes `Modelless Form` ;
- `Locale` : armazena os arquivos de internacionalização;
- `Middleware` : contém as classes *Middleware*;
- `Model` : contém as entidades da aplicação, os modelos;
- `Shell` : diretório para criação de tarefas;
- `View` : armazena as classes responsáveis pela apresentação e visualização das páginas.

É importante, no decorrer do desenvolvimento, assegurar que os arquivos criados estão nos locais definidos pela convenção do framework. Isso garante um bom funcionamento da aplicação e o código ficará de acordo com as normas do CakePHP, facilitando a manutenção e o entendimento quando outros desenvolvedores precisarem trabalhar no projeto.

PERMISSÕES NOS DIRETÓRIOS TMP E LOGS

Para os desenvolvedores, em especial os que utilizam UNIX, é preciso lembrar de dar permissões de escrita nos diretórios `tmp` e `logs`. Isso vai evitar que problemas apareçam durante o desenvolvimento do projeto ou quando for hospedar o projeto em algum servidor. Esses diretórios são utilizados pelo CakePHP para guardar vários tipos de arquivos, como dados de sessão, cache e arquivos temporários. Após criar o projeto, é sempre importante verificar as permissões desses diretórios, mas o próprio CakePHP faz essa verificação para alertar ao desenvolvedor e garantir o perfeito funcionamento. Usuários Windows, por padrão, não vão ter esse problema e não precisarão fazer configurações desse tipo.

2.3 Constantes

O CakePHP fornece um conjunto de constantes para representar os diretórios da aplicação. São variáveis bem úteis para a pessoa desenvolvedora quando ela pretender usar o nome de um diretório descrito na convenção do framework para compor alguma lógica. A lista de constantes para representar os diretórios é:

- `APP` : caminho absoluto para o diretório do aplicativo, incluindo uma barra final;
- `APP_DIR` : nome do diretório do aplicativo;
- `CACHE` : caminho para o diretório de arquivos em cache;
- `CAKE` : caminho para o diretório do projeto;
- `CAKE_CORE_INCLUDE_PATH` : caminho para o diretório `lib` do CakePHP;
- `CONFIG` : caminho para o diretório de configuração;
- `CORE_PATH` : caminho para o diretório CakePHP incluindo a barra final de diretório;

- `DS` : abreviação de `DIRECTORY_SEPARATOR` , cujo valor é `/` no Linux e `\\` no Windows;
- `LOGS` : caminho para o diretório de logs;
- `ROOT` : caminho para o diretório raiz;
- `TESTS` : caminho para o diretório de testes;
- `TMP` : caminho para o diretório de arquivos temporários.
- `WWW_ROOT` : caminho completo para o diretório web.

2.4 Testando a aplicação em localhost

Configurar um servidor HTTP é, por vezes, uma tarefa que requer tempo e conhecimento. Pensando em produtividade, o CakePHP vem com um servidor integrado, simples, porém eficiente, que vai permitir testar nossa aplicação no nosso ambiente local. Para inicializar o servidor, precisamos executar um comando dentro do diretório `bin` :

```
bin/cake server
```

Esse comando deixará a aplicação acessível no endereço `http://localhost:8765/` . Caso a porta `8765` ou o endereço `localhost` não esteja disponível, podemos passar o parâmetro **-H** para definir um novo endereço:

```
bin/cake server -H 192.168.10.100 -p 2222
```

Após a execução do comando, a aplicação passará a ficar disponível no endereço `http://192.168.10.100:2222/` . Simples, não é?

2.5 Criando a aplicação AdPET

Estamos prontos para iniciar a aplicação AdPET. Já abordamos os conceitos necessários para criar um projeto CakePHP e agora

precisamos praticar tudo o que aprendemos. Um passo importante a ser lembrado é que precisamos ter o Composer devidamente instalado na máquina. Assumindo que o ambiente de desenvolvimento contém os requisitos necessários, vamos criar o projeto com o comando:

```
composer create-project --prefer-dist cakephp/app:~4.0 AdPET
```

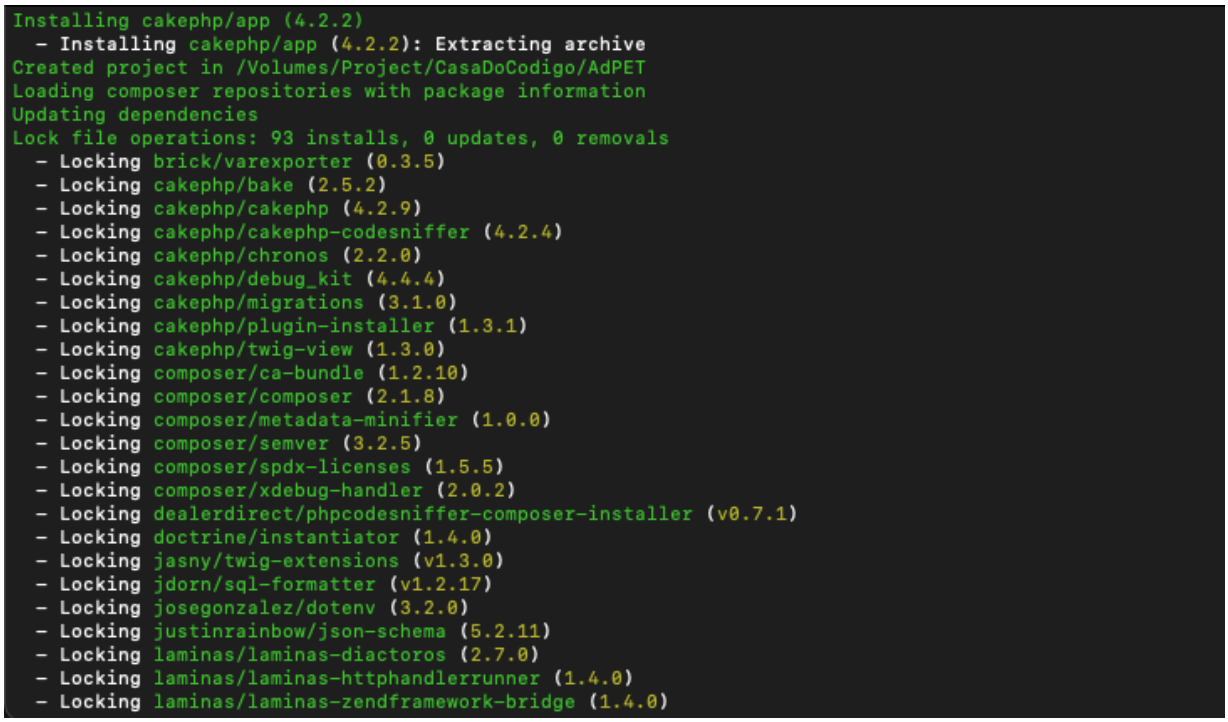


Figura 2.1: Executando comando para criar um projeto CakePHP.

No momento da criação do projeto, não é necessário especificar as configurações do um banco de dados que a aplicação vai utilizar, porém é preciso garantir que a versão do PHP instalada na máquina seja no mínimo a **7.2**. Para testar rapidamente se o projeto foi criado corretamente, inicializamos o servidor HTTP da aplicação executando o comando:

```
cake server
```

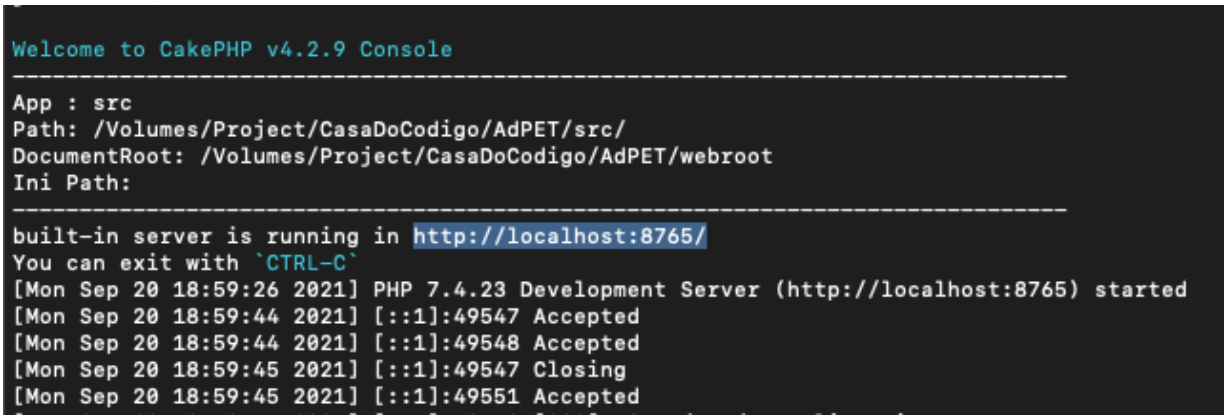


Figura 2.2: Executando comando para inicializar o servidor.

Se os passos descritos funcionarem corretamente, teremos a seguinte página, disponível no endereço `http://localhost:8765/`, visualizada no browser.

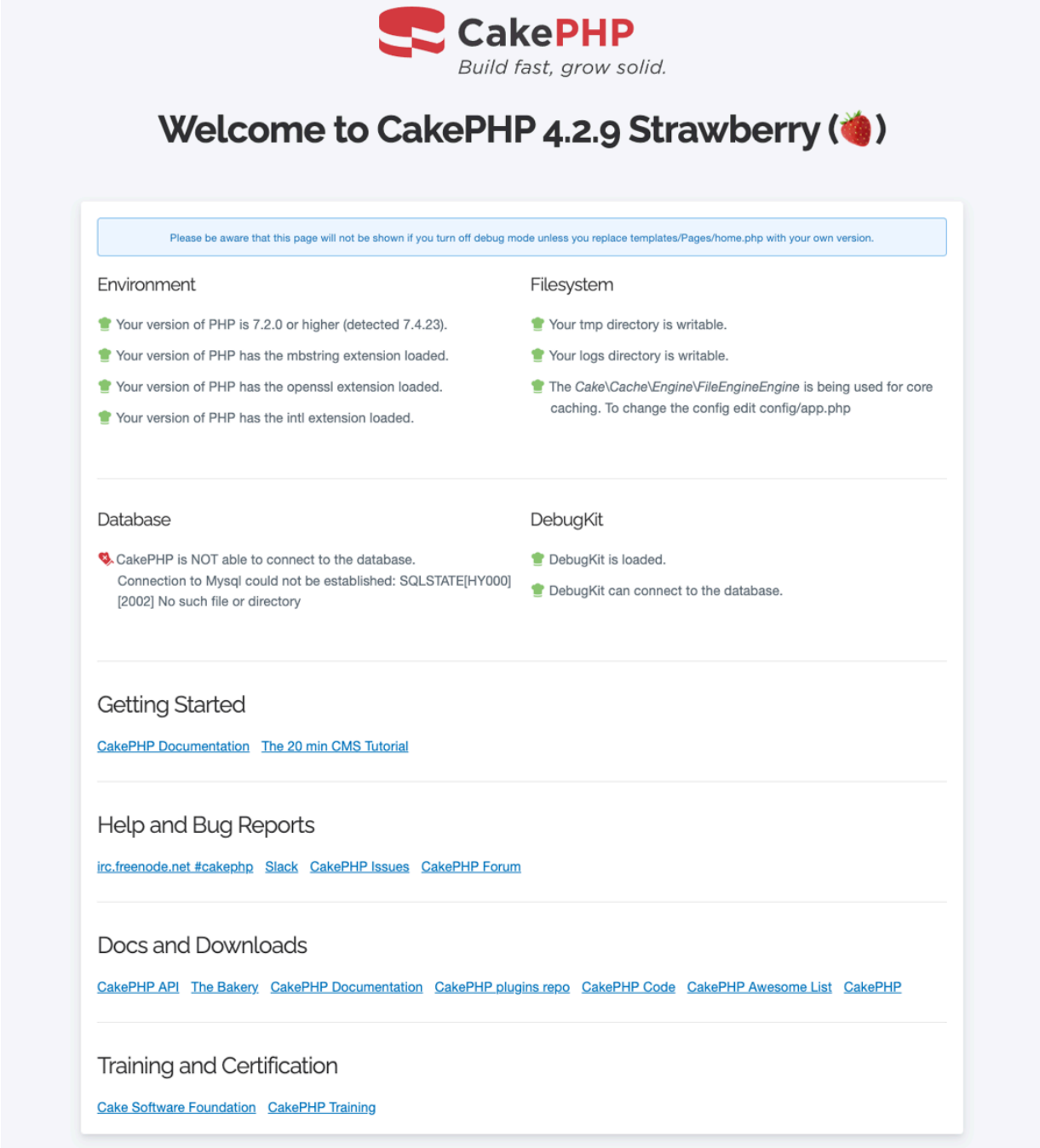


Figura 2.3: Tela inicial do CakePHP versão 4.2.9.

Quando o projeto foi criado, a versão atual do CakePHP era a 4.2.9, como observado na imagem anterior. Porém, durante seus estudos, é possível que a versão do CakePHP sofra alguma atualização, mas não se preocupe, pois isso não será um problema caso a versão 4 seja mantida. Mudanças de versão podem ser também correção de

problemas, atualizações de bibliotecas etc. Por exemplo, o projeto poderia também ser desenvolvido na versão 4.0.1, como exemplificado na imagem a seguir. A página inicial sofre leves mudanças de layout.

Figura 2.4: Tela inicial do CakePHP versão 4.0.1.

Um outro detalhe nesses dois casos que exemplifiquei anteriormente, usando diferente versões do framework, é que a estrutura do projeto se mantém igual. Toda a sua estrutura de diretório é baseada em um paradigma chamado de *CoC*, que estabelece uma árvore de diretórios e arquivos padronizados. Esse é um detalhe que considero importante na avaliação da escolha de um framework, pois essa padronização facilita e normaliza o conhecimento do projeto em uma equipe de desenvolvimento.

O que é convenção sobre configuração (CoC)?

Convenção sobre configuração (CoC, do inglês *Convention over Configuration*) visa simplificar algumas etapas do processo de

desenvolvimento. Seguindo algumas convenções adotadas, algumas definições são retiradas da responsabilidade do desenvolvedor, tais como a estruturação de diretórios e a configuração de dependências. Esse modelo facilita a padronização do código e melhora o entendimento e a produtividade em inicializar funcionalidades.

O CakePHP, assim como outros frameworks modernos, utiliza a CoC como modelo para desenvolvimento de software. Um dos pontos iniciais quando estamos utilizando um framework é saber quais são as convenções de código adotadas; por isso, em certos casos, costuma-se afirmar que a curva de aprendizado se torna lenta quando é decidido usar um framework em um projeto. Porém, passada a fase de aprendizado, o ganho de produtividade supera o modelo convencional adotado em projetos mais antigos. Neste livro, quando algum novo aspecto for introduzido, será apresentada a convenção adotada pelo CakePHP para que o código gerado obedeça e funcione como esperado.

Convenção para o nome das classes

A primeira convenção que quero apresentar é em relação ao nome das classes. Sempre que for preciso criar uma classe, a nomenclatura deve seguir o padrão *CamelCase*, no qual cada palavra deve ser iniciada com uma letra maiúscula e unidas sem espaços. O mesmo padrão deve ser adotado para o nome dos arquivos. Por exemplo, a classe chamada `DateUtil` deve ter o arquivo `DateUtil.php`, ou seja, o nome são duas palavras, `Date` e `Util`, iniciadas com letras maiúsculas e unidas.

A regra para atribuir nomes para as classes *controllers* deve seguir também o padrão CamelCase, mas flexionando a palavra para o plural e adicionando o sufixo "Controller". Por exemplo, veja os nomes `PetsController` e `UsersController`, que seguem o CamelCase e possuem as palavras `Pets` e `Users` no plural seguidas do sufixo "Controller". Esse tópico é apenas para apresentar essa primeira

convenção; vamos detalhar mais o conteúdo sobre Controllers no capítulo 6.

2.6 Configurando o CakePHP

Em qualquer aplicação, sempre será necessário configurar alguns detalhes peculiares do projeto que está em desenvolvimento. No CakePHP, há uma série de itens que podem ser modificados. Todos os principais arquivos de configuração estão localizados no diretório `config` :

- `app.php` : contém as propriedades gerais da aplicação, como *namespace*, configuração para o banco de dados, entre outras. É um arquivo que certamente será modificado pelo desenvolvedor;
- `app_local.php` : arquivo de configuração local para sobrescrever a sua configuração em `app.php` ;
- `bootstrap.php` : possui um grupo de configuração mais relacionado às dependências do framework, configuração de plugins, componentes etc.;
- `bootstrap_cli.php` : usado para adicionar configurações relacionadas aos diferentes ambientes de desenvolvimento que a aplicação poderá utilizar;
- `paths.php` : arquivo usado para adicionar constantes relacionadas a caminhos de arquivos ou diretórios — por exemplo, o *path* do diretório de upload;
- `requirements.php` : arquivo responsável por declarar todos os requisitos requeridos para o funcionamento do projeto;
- `routes.php` : contém as rotas da aplicação.

O `app.php` é o principal arquivo para definir as propriedades comuns para o projeto e para os ambientes em que a aplicação vai ser executada. Dentre os itens do `app.php` , podemos destacar a

propriedade `App`, que é um *array* de configurações. Algumas propriedades que considero importantes para o aprendizado são:

- `namespace`: define o *namespace* principal das classes do projeto;
- `baseUrl`: configura o CakePHP para não usar o modo `mod_rewrite`;
- `base`: é o diretório raiz do projeto, caso seja setado com valor *false*; caso contrário, deve ser passada uma string iniciando com o caractere `/`;
- `defaultLocale`: define o *locale* da aplicação, por exemplo, `pt_BR`, `en_GB` etc.;
- `dir`: diretório para armazenar o código-fonte da aplicação;
- `encoding`: define a codificação dos caracteres utilizados nos arquivos HTML e também para as conexões com o banco de dados — normalmente, utiliza-se a codificação UTF-8;
- `webroot`: diretório público da aplicação. Contém arquivos JS, CSS, imagens, entre outros elementos;
- `fullBaseUrl`: configuração usada para definir URLs absolutas, úteis para quando não for possível acessar a variável `$_SERVER`;
- `imageBaseUrl`: define o diretório dos arquivos das imagens do projeto. Está abaixo do diretório *webroot*;
- `cssBaseUrl`: define o diretório dos arquivos `CSS` do projeto. Está abaixo do diretório *webroot*;
- `jsBaseUrl`: define o diretório dos arquivos `JavaScript` do projeto. Está abaixo do diretório *webroot*.

O QUE SÃO NAMESPACES

Namespaces são declarações que agrupam classes, constantes, interfaces ou funções sob um mesmo contexto, melhorando a organização do código. Esse recurso foi adicionado somente a partir da versão 5.3 do PHP e, quando corretamente utilizado, impede colisões entre nomes de classes, funções ou interfaces.

O princípio é simples: artefatos com o mesmo nome não podem existir usando o mesmo namespace, mas nada impede que, por exemplo, o mesmo nome de uma classe exista em diferentes namespaces. Fazendo uma analogia com diretórios de sistemas operacionais, arquivos com o mesmo nome não podem existir no mesmo diretório, mas o podem se os diretórios forem diferentes.

No arquivo `app.php`, em geral, podemos definir configurações como:

- Acesso ao banco de dados;
- Cache;
- E-mail;
- Gerenciamento de erros;
- Configuração de sessão.

Vale ressaltar que essas configurações se aplicam a todo o projeto e ambientes. Algumas propriedades definidas em `app.php` serão sobrescritas com a configuração feita em `app_local.php`, como a configuração do banco de dados para o desenvolvimento local. É importante entender essa hierarquia entre esses arquivos para saber exatamente onde escrevemos as configurações para um perfeito funcionamento do framework.

2.7 Configurando o banco de dados

Você deve ter notado que, quando visualizamos a aplicação no browser, a página inicial informa quais itens estão de acordo com o *checklist* inicial do CakePHP. O item `Database` vai indicar um problema, o que é um erro óbvio, já que até agora apenas criamos a estrutura padrão do projeto usando `create-project`. Ainda precisamos definir as informações que permitem acesso ao banco de dados.

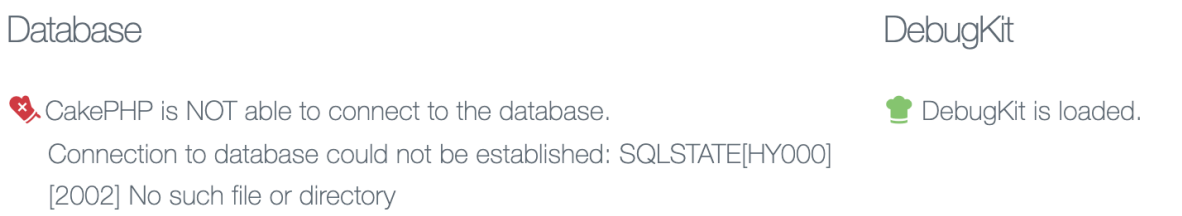


Figura 2.5: Erro na inicialização do banco de dados.

Por padrão, o banco de dados com o qual o CakePHP já vem previamente configurado quando criamos o projeto é o MySQL. Este é o gerenciador que vamos utilizar no desenvolvimento da aplicação AdPET. O CakePHP pode trabalhar com outros bancos de dados caso você tenha outra preferência, como:

- PostgreSQL 8.3+;
- Microsoft SQL Server (2008 ou superior);
- SQLite 3.

A configuração do banco de dados será definida no arquivo `app_local.php`, localizado no diretório `config`, na propriedade `Datasources`, que é um basicamente um `array`. Nessa propriedade, podemos definir várias conexões diferentes de acesso a bancos de dados, de acordo com a necessidade do projeto. Cada conexão dentro do `Datasources` é um outro *array* com informações para conectar ao banco de dados.

```
'Datasources' => [
        'default' => [],
```

```
        'conexao_01' => [],
        'conexap_02' => []
]
```

As principais opções que utilizaremos para configurar uma conexão com o banco de dados são:

- `host` : endereço de acesso ao banco de dados;
- `username` : nome do usuário com acesso ao banco de dados;
- `password` : senha do usuário;
- `database` : nome do banco de dados;
- `port` : número da porta de acesso. Essa é uma configuração opcional, o CakePHP utiliza a porta padrão de cada gerenciador.

Existem outras propriedades pertinentes ao banco de dados que devem ser configuradas no arquivo `app.php` . Por exemplo, no projeto AdPET, assumimos que o banco de dados é o `MySQL` ; então, configurações como o *Driver* e o *encoding* também devem ser indicadas no arquivo `app.php` . As outras propriedades que podemos também definir são:

- `className` : namespace da classe responsável por gerenciar a conexão com o banco de dados;
- `driver` : driver para trabalhar com o banco de dados;
- `persistent` : possibilita usar ou não uma conexão persistente com o banco de dados;
- `encoding` : codificação para as transações com o banco de dados;
- `timezone` : *time zone* (fuso horário) do servidor;
- `log` : habilita o log de consulta;
- `cacheMetadata` : habilita o uso de cache para metadados.

A conexão principal utilizada pelo CakePHP para interagir com o banco de dados é chamada `default` ; por essa razão, é uma boa prática deixá-la configurada. A seguir, um exemplo de uma configuração usando as opções descritas anteriormente para definir a conexão `default` utilizando o MySQL:

```
'Datasources' => [
    'default' => [
        'host' => 'localhost',
        'username' => 'meu_usuario',
        'password' => 'minha_senha',
        'database' => 'meu_banco_de_dados'
    ],
```

Essas opções se repetem quando é necessário configurar outras conexões em que o sistema vai trabalhar. No nosso caso, só configurar uma conexão `default` já é suficiente. Para projetos maiores, é uma boa prática ter uma conexão configurada para os ambientes em que a aplicação vai ser executada. Por exemplo, podemos utilizar a conexão `default` para o desenvolvimento e criar outra para ser utilizada quando a aplicação estiver em produção ou quando for executar testes.

```
'Datasources' => [
     'default' => [
        'host' => 'localhost',
        'username' => 'meu_usuario',
        'password' => 'minha_senha',
        'database' => 'meu_banco_de_dados'],
    'test' => [
        'host' => 'localhost',
        'username' => 'meu_usuario_de_teste',
        'password' => 'minha_senha_de_teste',
        'database' => 'meu_banco_de_dados_de_teste'],
```

Gerenciando conexões manualmente

Um artefato importante que o CakePHP tem em seu conjunto quando estamos configurando banco de dados é a classe `Cake\Datasource\ConnectionManager`. Essa classe possibilita que, em tempo de execução, seja configurada uma nova conexão com um banco de dados através do método `config()`. Por exemplo, caso seja

preciso em uma determinada situação abrir uma conexão com um banco de dados externo, podemos criar o seguinte código:

```
use Cake\Datasource\ConnectionManager;

ConnectionManager::config('QA', [
    'className' => 'Cake\Database\Connection',
    'driver' => 'Cake\Database\Driver\Mysql',
    'persistent' => false,
    'host' => 'localhost',
    'username' => 'my_app',
    'password' => 'sekret',
    'database' => 'my_app',
    'encoding' => 'utf8',
    'timezone' => 'UTC',
    'cacheMetadata' => true,
]);
```

Esse trecho de código define os parâmetros da conexão `QA` sem precisar defini-los no arquivo `app.php` . Outra funcionalidade da classe `ConnectionManager` é inicializar uma conexão, porém já definida em `Datasources` com o método `get()` . Por exemplo, para carregar a conexão *production* definida no arquivo `app.php` , poderíamos escrever:

```
use Cake\Datasource\ConnectionManager;

$conn = ConnectionManager::get('production');
```

Esse código cria o objeto `$conn` , que é uma instância da classe `Cake\Datasource\ConnectionInterface` . Com esse objeto, podemos executar operações SQL, por exemplo:

```
use Cake\Datasource\ConnectionManager;

$conn = ConnectionManager::get('production');
$results = $conn->execute('SELECT * FROM pets')->fetchAll('assoc');
```

Nesse trecho de código, estamos abrindo uma conexão com o banco de produção e executando uma consulta SQL na tabela `pets`. Nesse caso, é uma `select` simples que retorna todos os registros desta tabela. É um recurso valioso quando temos que fazer alguma integração com um banco de dados externo. Mais detalhes sobre essa classe podem ser lidos em: `https://book.cakephp.org/4/en/orm/database-basics.html#Cake\Datasource\ConnectionManager`.

2.8 Configurando o banco de dados do AdPET

Voltando novamente para o projeto AdPET, é necessário definir o acesso ao banco de dados do sistema, pois vamos precisar dessa etapa devidamente configurada para prosseguir com outros tópicos nos capítulos seguintes. Levo em consideração que o leitor ou leitora tenha conhecimentos básicos sobre banco de dados e que saiba configurar os primeiros passos — como criar um banco de dados e garantir acesso aos usuários.

Embora o livro trabalhe com o MySQL, nada impede que você use o gerenciador de banco de dados de sua preferência, seja o PostgreSQL ou outro. Só não se esqueça de utilizar o driver correto caso mude para outra opção. Nesse primeiro momento, precisamos somente editar a configuração `default` em `Datasources` no arquivo `config/app.php` e em `config/app_local.php`:

```
    'Datasources' => [
        'default' => [
            'className' => Connection::class,
            'driver' => Postgres::class,
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'user_adpet',
            'password' => '123456',
```

```
            'database' => 'adpet',
            'encoding' => 'utf8',
            'timezone' => 'UTC',
            'cacheMetadata' => true,
            'log' => false
        ],
```

Uma observação importante: é preciso dar permissão ao usuário para manipular tabelas, ou então será impossível para o CakePHP gerar as modificações necessárias nelas. Para testar rapidamente se a configuração foi feita corretamente, podemos subir a aplicação com o comando `bin/cake server`. Se as informações editadas na propriedade `Datasources` no arquivo `app_local.php` forem válidas, é possível visualizar uma mensagem informando que o banco está configurado corretamente. Se o servidor HTTP estava ativo antes das modificações, é preciso apenas dar um refresh no browser.

Figura 2.6: Mensagem de confirmação na página inicial.

Recomendo o uso da ferramenta `phpMyAdmin` como opção para trabalhar com o MySQL. Com ela, podemos criar diversos bancos de dados e manipular colunas e registros usando uma interface web bem prática. O `phpMyAdmin` precisa do ambiente de desenvolvimento com o Apache e o PHP instalados e configurados. Caso prefira, nada impede que você gerencie também o MySQL por um terminal de comando.

Conclusão

Aprendemos a instalar o CakePHP e a fazer as primeiras configurações necessárias para inicializar o desenvolvimento do projeto. É essencial ter certeza de que o banco MySQL está perfeitamente instalado e o acesso configurado corretamente no arquivo `app.php` e em `app_local.php`. No capítulo seguinte, vamos aprender como criar a estrutura do banco de dados do projeto AdPET usando *migrations*.

Capítulo 3
Migrations

Com o CakePHP devidamente instalado e as configurações básicas criadas, podemos dar continuidade ao aprendizado. Vamos criar agora a estrutura do banco de dados do projeto, uma vez que já configuramos o acesso a ele (um dos objetos de estudo do capítulo anterior).

É usual a necessidade de usar um banco de dados para manter armazenadas informações como dados dos usuários, mensagens, arquivos, entre outros dados relevantes para o sistema. No processo de desenvolvimento de software, à medida que o sistema cresce, é importante manter a sincronia entre o código desenvolvido e o banco de dados para evitar problemas futuros. Na aplicação AdPET, serão criadas tabelas para armazenar informações sobre os animais, os usuários do sistema, as adoções, entre outras.

Atualmente, conseguimos realizar mudanças na estrutura de um banco de dados usando *migrations*, que são um conjunto de classes que executam operações como criação de tabelas, adição de colunas, inserção de dados — ou seja, operações que seriam feitas diretamente no banco de dados usando SQL padrão. As migrations, além de controlar as alterações, permitem manter o histórico das evoluções ocorridas, como acontece com os arquivos usando algum sistema de controle de versão, por exemplo, o Git. Com isso, agregamos mais valor no processo de desenvolvimento, pois adicionamos mais controle nas mãos de quem desenvolve. O uso de migrations oferece para a pessoa programadora uma forma de espelhar a estrutura do banco de dados em linhas de códigos, e isso é sensacional.

Após a fase de instalação do framework e dos requisitos necessários para o desenvolvimento, iniciar a construção do software com migrations é mais que recomendado. Usar essa metodologia, por

menor que seja o sistema, vai incentivar quem desenvolve a fazer uma análise inicial a fim de identificar as entidades envolvidas, como as informações serão salvas na estrutura do banco de dados e como vai ser a estrutura de relacionamento entre os dados.

3.1 Criando uma migration

O CakePHP tem suporte nativo para trabalhar com *migrations* usando a API desenvolvida pela Phinx (disponível em https://book.cakephp.org/phinx/0/en/migrations.html). Para criar uma migration usando `bake`, temos duas opções de comandos:

```
bin/cake bake migration NomeDoArquivo
...
bin/cake migrations [acao] NomeDoArquivo
```

Ou:

```
bin/cake bake migration NomeDoArquivo nome:string created
```

Perceba a diferença entre os dois comandos: o primeiro é recomendado para criar arquivos vazios com apenas o esqueleto da classe, enquanto o segundo pode ser usado para passar uma relação de colunas que serão criadas para a tabela, o que acelera a escrita e desenvolvimento do código.

O CakePHP recomenda que os nomes das migrations sigam o formato CamelCase. Também é recomendado que o nome contenha a indicação da ação que a migration vai executar — por exemplo: `20190428195255_create_pets.php` e `20190428195255_add_description_pets.php`. A primeira migration informa que a tabela `pets` será criada; a segunda informa que será adicionada uma coluna nova na tabela.

Por uma questão de praticidade, eu recomendo utilizar a sintaxe do primeiro comando. Não é obrigatório passar na execução do

comando a lista de colunas. Caso seja informado apenas o nome, o arquivo vai ser criado sem nenhum problema.

Quando o comando finaliza sua execução, um arquivo será criado com o nome no seguinte formato: `YYYYMMDDHHMMSS_create_nome_da_tabela.php`. A utilização de datas no início do nome dos arquivos determina a ordem da execução das migrations de forma crescente, a partir da mais antiga para a mais recente. Todos os arquivos de migração do banco de dados são criados no diretório `config/Migrations`.

Para um melhor aprendizado, recomendo que você pratique as duas maneiras e perceba as diferenças na criação dos arquivos, por exemplo, passando colunas como parâmetros. O esqueleto de uma classe vazia segue o modelo:

```
<?php
use Migrations\AbstractMigration;

class NomeDoArquivo extends AbstractMigration
{
    /**
     * Change Method.
     * @return void
     */
    public function change()
    {
    }
}
```

Observando a estrutura, a classe possui um único método, `change()`. Em uma migration, temos três principais métodos que podemos usar para manipular uma tabela no banco de dados:

- `change()`
- `up()`
- `down()`

A partir da versão 0.2.0 da Phinx, o conceito *reversible migrations* foi implementado, tornando `change()` o método principal da classe `AbstractMigration`. Com esse conceito, apenas precisamos definir a lógica que cria a estrutura de uma tabela; o framework vai automaticamente saber como reverter as alterações no banco de dados, se preciso. Caso o método `change()` seja definido, automaticamente os métodos `up()` e `down()` serão ignorados; logo, se for preciso utilizar esses métodos legados, é preciso criar um arquivo de migração que não contenha o `change()`.

O método `up` vai ser necessário quando modificarmos o banco de dados e detectarmos que a migração em questão não foi executada anteriormente. Já o `down()` é utilizado para reverter as transformações descritas no método `up`. Em versões antigas, o `up()` era o principal método responsável por conter a lógica necessária para criar tabelas, colunas, entre outras operações.

O QUE É O BAKE?

Bake é uma ferramenta para linha de comando criada para o CakePHP que permite criar, de forma fácil e rápida, estruturas básicas como *models*, *controllers* e outros tipos de estruturas utilizadas pelo framework. Sempre que possível, nos exemplos deste livro, vamos executar um comando `bake` para exemplificar o seu uso e como utilizá-lo para criar um determinado esqueleto de classe do CakePHP.

3.2 Criando as tabelas do projeto

Vamos iniciar com a criação de algumas tabelas para o projeto. Embora seja possível usar uma migration e criar toda a estrutura do banco, é recomendado criar uma migration para cada tabela; assim, organizamos melhor o controle do banco de dados.

O banco de dados do AdPET será formado por nove tabelas: `states`, `cities`, `addresses`, `families`, `breeds`, `users`, `pets`, `adoptions` e `comments`.

> Nomes das tabelas no CakePHP
>
> O CakePHP define uma convenção para os nomes das tabelas. O nome deve estar no plural e, se forem nomes compostos, as palavras precisam ser separadas pelo caractere *underline* (_) — por exemplo, `pets`, `users`, `20190428195255_create_pets.php`, `breeds`. O idioma padrão usado para nomear as tabelas ou colunas é o inglês. Ao escrever palavras com o idioma padrão, o CakePHP consegue determinar o singular de uma palavra no plural ou vice-versa. Caso pretenda definir outro idioma fora da convenção, a classe `Cake\Utility\Inflector` deve ser utilizada para criar novas regras idiomáticas. Para mais detalhes sobre a classe `Inflector`, acesse: `https://book.cakephp.org/4.0/en/core-libraries/inflector.html#Cake\Utility\Inflector`.

As primeiras migrations serão para as tabelas `states`, `cities` e `addresses`. São tabelas que criamos para organizar o endereço dos usuários do sistema. Para a tabela `states`, temos:

```
$ bin/cake bake migration createStates
```

O comando vai gerar uma migration com algum código e deixará para a pessoa desenvolvedora continuar a lógica, adicionar colunas, índices e outras estruturas.

```
<?php
use Migrations\AbstractMigration;

class NomeDoArquivo extends AbstractMigration
{
    /**
     * Change Method.
     * @return void
```

```
     */
    public function change()
    {
        $table = $this->table('states');
        $table->create();
    }
}
```

Como a ideia central do CakePHP é a agilidade, podemos também passar parâmetros indicando quais colunas queremos adicionar à migration. Em vez de criar uma classe somente com a definição da tabela, já teríamos uma classe criada com código para criar tabela e colunas. O formato dos parâmetros que devemos passar para criar as colunas da tabela deve seguir este modelo:

```
NomeDoCampo:TipoDoCampo[tamanho]:TipoDoIndice:NomeDoIndice
```

Para as tabelas `cities` e `addresses`, o comando que informa as colunas seria:

```
$ bin/cake bake migration createCities name:string
$ bin/cake bake migration createAddresses street:name zipcode:integer
```

Os códigos finais para essas migrations são:

```
<?php
use Migrations\AbstractMigration;

class CreateStates extends AbstractMigration
{

    public function change()
    {
        if (!$this->hasTable('states')){
            $table = $this->table('states');

            $table->addColumn('name', 'string', [
                'limit' => 255,
                'null' => false,
            ]);
```

```
            $table->create();
        }
    }
```

```
<?php

use Migrations\AbstractMigration;

class CreateCities extends AbstractMigration
{
    public function up()
    {

        if (!$this->hasTable('cities')) {
            $table = $this->table('cities');
            $table->addColumn('name', 'string', [
                'limit' => 255,
                'null' => false
            ]);
            $table->create();
        }
    }
}
```

```
<?php
use Migrations\AbstractMigration;

class CreateAddresses extends AbstractMigration
{

    public function change()
    {

        if (!$this->hasTable('addresses')) {
            $table = $this->table('addresses');

            $table->addColumn('street', 'string', [
                'limit' => 255,
                'null' => false
```

```
            ]);

            $table->addColumn('zip_code', 'integer', [
                'default' => null,
                'limit' => 255
            ]);
            $table->create();
        }
    }
}
```

> Chave primária
>
> Não é preciso informar a coluna **id** no código da migration. O CakePHP vai criar essa coluna na tabela, que será autoincremental e do tipo `integer`.

Nas classes exibidas anteriormente, dentro do método `change()` temos este trecho: `$table = $this->table('$nome_da_tabela')`. Esse método recebe como parâmetro o nome da tabela e retorna um objeto que chamamos de `$table`, o qual é utilizado para adicionar colunas. Ele também pode ser usado para criar índices, chaves estrangeiras etc. Sem esse objeto inicializado corretamente com o nome da tabela, a migration não vai funcionar.

O método `addColumn()` é responsável por criar as colunas da tabela e recebe como parâmetros o nome, que é obrigatório, e o tipo de dado. Finalizando, o método `create()` aplica no banco de dados tudo o que foi codificado na migration. Uma observação importante é que, ao final do `change()`, não podemos esquecer de executar o método `create()` para que as mudanças sejam, de fato, aplicadas no banco de dados.

Quando estamos trabalhando com o `up()` e o `down()`, podemos usar outro método, chamado `save()`, para persistir as mudanças, como no exemplo demonstrado a seguir:

```
class createTests extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $users = $this->table('tests');
        $users->addColumn('username', 'string',
                          array('limit' => 20))
              ->addColumn('password', 'string',
                          array('limit' => 40))
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {
        $this->execute('DELETE FROM tests');
    }
}
```

O exemplo foi citado apenas para apresentar outra maneira de trabalhar com Phinx. Aqui, vamos sempre escrever o código das nossas migrations no método padrão `change()`.

A API Phinx pode automaticamente reverter as alterações feitas no banco de dados quando usamos em nossas migrations algum dos métodos listados a seguir:

- `createTable()` : cria uma tabela;
- `renameTable()` : renomeia a tabela;
- `addColumn()` : adiciona uma coluna na tabela;
- `renameColumn()` : renomeia a coluna;
- `addIndex()` : adiciona index na tabela;
- `addForeignKey()` : adiciona chaves estrangeiras.

Caso não usemos esses métodos para manipular a criação da estrutura das tabelas, ou algum outro problema ocorra durante a reversão, a migration lançará uma exception do tipo `IrreversibleMigrationException`.

3.3 Tipos de dados das colunas

Agora, precisamos agregar mais conhecimento para trabalhar com as migrations. O primeiro ponto que quero apresentar são as opções que a Phinx oferece para representar nossos dados. O banco de dados precisa armazenar informações de todos os tipos e formatos, sejam elas textos ou objetos mais complexos, como `blob`. Com o método `addColumn()`, podemos criar colunas com diferentes tipos de dados apenas especificando o formato no segundo parâmetro do método.

Os tipos de dados suportados são: `string`, `text`, `integer`, `float`, `biginteger`, `decimal`, `datetime`, `timestamp`, `time`, `date`, `binary`, `boolean` e `uuid`.

Claro que esses tipos não são exatamente os que utilizamos no banco de dados para determinar que tipo de dado uma coluna da tabela vai armazenar. Porém, quando estamos escrevendo uma migration, por exemplo, usamos um tipo `string`, que, na verdade, vai representar uma coluna do tipo `VARCHAR`. Ambos os tipos servem para armazenar dados do tipo texto. Para fixar melhor o conteúdo, veja mais alguns exemplos para criar colunas:

```
$table->addColumn('name','string');
  $table->addColumn('data_de_nascimento','timestamp');
  $table->addColumn('ativo','boolean');
```

Quando estamos usando o MySQL, podemos ainda acrescentar os tipos `enum`, `set`, `blob` e `json` se a versão do MySQL for superior à **5.7**. Para o PostgreSQL, nas versões acima de 9.3, é possível criar

com os tipos `smallint`, `json`, `jsonb` e `uuid`. Caso não especifiquemos o tipo de dado, a migration usa, por padrão, o tipo `string` para definir o tipo da coluna, como no trecho a seguir:

```
$table->addColumn('name');
```

Definindo opções nas colunas

Em algumas colunas, precisamos adicionar opções extras que definem validações para os dados que a tabela vai receber, como aceitar valores nulos e limite de caracteres. Para definir um grupo de opções para a coluna, devemos criar um `array` contendo todas as opções necessárias. Este dado é informado no terceiro parâmetro do método `addColumn()`. As opções padrões que podem ser utilizadas para qualquer tipo de coluna são:

- `limit`: determina o número máximo de caracteres;
- `default`: determina o valor padrão;
- `null`: permite valores nulos ou não;
- `after`: especifica a posição que deve ser inserida em relação a outra coluna;
- `comment`: adiciona um comentário para a coluna.

Nas migrations criadas anteriormente, podemos ter o seguinte código:

```
$table->addColumn('name', 'string',
[
    'limit' => 255,
    'null' => false,
]);
```

Com esse código, estamos informando que a coluna deve ser criada para não aceitar valores nulos (`null => false`) e com o limite de 255 caracteres (`limit => 255`). Para cada tipo de dado, existem opções específicas que podemos passar junto do método `addColumn()`. Para cada tipo, as principais configurações são:

Integer:

- `identity` : habilita ou desabilita o autoincremento;
- `signed` : para o MySQL, habilita ou desabilita a opção `unsigned` .

```
$table->addColumn('exemplo', 'integer',
    array(
        'identity' => true,
         'signed' => true
    ));
```

Boolean:

- `signed` : para o MySQL, habilita ou desabilita a opção `unsigned` .

```
$table->addColumn('exemplo', 'boolean',
    array('signed' => true));
```

Decimal:

- `precision` : deve ser combinado com a escala para definir a precisão decimal;
- `scale` : combinado com o valor da opção precisão, define a precisão decimal;
- `signed` : para o MySQL, habilita ou desabilita a opção `unsigned` .

```
$table->addColumn('exemplo', 'decimal',
    array(
        'precision' => 5,
        'scale' => 2,
        'signed' => true
    ));
```

Timestamp:

- `update` : definir uma *trigger* quando o valor for atualizado;
- `timezone` : ativar ou desativar o fuso horário para o banco de dados PostgreSQL.

```
$table->addColumn('exemplo', 'timestamp',
    array(
```

```
            'update' => 'myTrigger',
            'timezone' => true
        ));
```

Enum ou **Set**:

- `values` : define uma lista de valores, separados por vírgula.

```
$table->addColumn('exemplo', 'enum',
    array(
        'values' => 'caju, manga, graviola'
    ));
```

Criação de novas tabelas para o AdPET

Vamos voltar ao nosso projeto e criar mais duas migrations, `families` e `breeds`, que serão necessárias para armazenar as informações da família e da raça dos pets, respectivamente. O comando para criar as migrations será:

```
$ bin/cake bake migration createFamilies name:string[255]
$ bin/cake bake migration createBreeds street:name[255]
```

A única diferença para os comandos já citados é o numeral **255**. Nesse caso, estamos informando o valor da propriedade `limit` para o campo `name` na tabela. Os resultados das migrations são:

```
<?php
use Migrations\AbstractMigration;

class CreateFamilies extends AbstractMigration
{
    public function change()
    {
        if (!$this->hasTable('families')) {
            $table = $this->table('families');

            $table->addColumn('name', 'string', [
                'default' => null,
```

```
                'limit' => 255,
                'null' => false,
            ]);

            $table->create();
        }
    }
}
```

```
<?php

use Migrations\AbstractMigration;

class CreateBreeds extends AbstractMigration
{

    public function change()
    {
        if (!$this->hasTable('breeds')) {
            $table = $this->table('breeds');

            $table->addColumn('name', 'string', [
                'default' => null,
                'limit' => 255,
                'null' => false,
            ]);

            $table->create();
        }

    }
}
```

3.4 Executando as migrations

Agora vamos testar tudo o que foi programado até agora. Para executar as migrations, devemos executar o seguinte comando:

```
    bin/cake migrations migrate
```

A saída no console deve ser algo como o texto a seguir:

```
$ bin/cake migrations migrate

using environment default
using adapter mysql
using database adpet

 == 20190417000001 CreateStates: migrating
 == 20190417000001 CreateStates: migrated 1.7270s

 == 20190417000002 CreateCities: migrating
 == 20190417000002 CreateCities: migrated 2.1396s

 == 20190417000003 CreateAddresses: migrating
 == 20190417000003 CreateAddresses: migrated 1.3793s

 == 20190417000004 CreateFamilies: migrating
 == 20190417000004 CreateFamilies: migrated 1.0274s

 == 20190417000005 CreateBreeds: migrating
 == 20190417000005 CreateBreeds: migrated 1.2826s


 All Done. Took 14.5790s
```

Com essa saída no console, as tabelas foram criadas com sucesso. Por padrão, o CakePHP vai usar o banco de dados configurado na conexão *default*, mas podemos também especificar uma conexão diferente onde as migrations serão aplicadas. É muito comum, por exemplo, em um determinado momento, precisar rodar os arquivos no banco de produção ou de teste. Com esse intuito, podemos passar a conexão como parâmetro `-c` na linha do comando, por exemplo:

```
bin/cake migrations migrate -c production
```

Esse comando vai aplicar as mesmas migrations no banco definido como *production* em vez do banco definido na conexão *default* .

3.5 Alterando tabelas

Quando executamos um grupo de migrations, tudo o que foi programado até o momento vai ser refletido no banco de dados. No decorrer do desenvolvimento de software, a probabilidade de que apareçam mudanças no sistema é uma realidade comum para os projetos. Mudanças podem ocorrer tanto no código-fonte como também precisam ser aplicadas no banco de dados.

As mudanças relacionadas ao banco de dados devem ser administradas também por migrations para fazerem as alterações nas tabelas. É recomendado que, para cada nova mudança, seja criada uma nova migration; assim, ao longo do desenvolvimento do sistema, mantemos o histórico da evolução do banco de dados.

Quando ainda estamos no ambiente de desenvolvimento e o projeto não foi disponibilizado em produção, é possível apagar o banco e recriá-lo totalmente reescrevendo novas migrations. Isso não será possível, ou pelo menos não é uma boa prática, se o sistema já tiver sido publicado para o uso final. Por isso, na fase inicial, é preciso modelar a estrutura de dados do sistema com cuidado.

Também não tem nenhum problema criar migrations sempre que necessário, mesmo que a aplicação ainda esteja na fase de desenvolvimento, pois assim criamos um bom hábito de programação e evitamos que, na fase final, quando o sistema vai ser homologado, nos esqueçamos de algo que vai quebrar no ambiente de produção.

O CakePHP tem uma convenção para a nomenclatura das migrations que vão ser usadas para aplicar mudanças em tabelas já existentes. As convenções para os nomes dessas migrations são:

- `(/^(Drop)(.*)/)` : quando a intenção é deletar uma tabela;
- `(/^(Add).*(?:To)(.*)/)` ou `(/^(Alter)(.\*)/)` : para adicionar novos campos à tabela;
- `(/^(Remove).*(?:From)(.*)/)` : para remover campos de uma tabela.

Vale lembrar que o desenvolvedor é livre para criar uma migration vazia e adicionar a lógica necessária. Basta seguir o formato `YYYYMMDDHHMMSS_NomeDaMigration` , que ela será executada sem nenhum problema. Porém, a organização é uma caraterística de um software com qualidade e, seguindo as convenções do framework, certamente o produto final será um software de qualidade.

Adicionando novas colunas

Vamos praticar um pouco com a tabela `addresses` para adicionar uma nova coluna e exemplificar a convenção adotada para adicionar algum novo campo em uma tabela. Nessa tabela, não temos uma coluna para guardar o número do endereço, então vamos adicionar essa coluna. Logo, o que precisamos agora é gerar uma nova *migration* para, de fato, aplicar essa mudança. O nosso comando vai ser:

```
bin/cake bake migration AddNumberToAddresses number:integer
```

O comando vai gerar uma migration como o nome `YYYYMMDDHHMMSS_AddNumberToAddresses.php` . Podemos observar que o nome utilizado no arquivo indica qual operação vai ser executada nessa migration e, automaticamente, o CakePHP vai gerar o código:

```
<?php
use Migrations\AbstractMigration;
```

```
class AddNumberToAddresses extends AbstractMigration
{

    public function change()
    {
        $table = $this->table('addresses');
        $table->addColumn('number', 'integer', [
            'default' => null,
            'limit' => 11,
            'null' => false,
        ]);
        $table->update();
    }
}
```

Se observarmos o diretório `config/Migrations`, até o momento, temos duas migrations relacionadas com a tabela `addresses`, `YYYYMMDDHHMMSS_CreateAddresses.php` e `YYYYMMDDHHMMSS_AddNumberToAddresses.php` — ou seja, uma migration que cria a tabela e outra que criamos agora para alterar a estrutura da mesma tabela adicionando uma nova coluna. Nessa nova migration, vale observar que o método usado para aplicar as mudanças é o `update()` e não mais o `create()`.

Removendo colunas

Para remover uma coluna, a convenção para escrever o comando tem que seguir o formato `(/^(Remove).*(?:From)(.*)/)`. Por exemplo, se for preciso remover a coluna recém-adicionada `number`, poderíamos escrever o seguinte comando:

```
bin/cake bake migration RemoveNumberFromAddresses number
```

O CakePHP vai gerar a seguinte migration:

```
<?php
  use Migrations\AbstractMigration;
```

```php
class RemoveNumberFromAddresses extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('addresses');
        $table->removeColumn('number');
    }
}
```

Note que na classe escrita não é preciso chamar o método `update()`. Esse exemplo não será usado na construção do projeto AdPET, foi citado apenas para aprendermos mais sobre como adicionar ou remover uma coluna da tabela.

3.6 Chaves estrangeiras e índices

O CakePHP trabalha com banco de dados relacionais, que são bancos que lidam com informações organizadas em tabelas, uma estrutura simples de linhas e colunas. As tabelas podem se relacionar entre si associando seus atributos com os atributos de outra tabela. Por exemplo, criamos as tabelas `states` e `cities` para representar os estados e as cidades no sistema AdPET. Podemos criar atributos que associem as informações entre elas. A regra para associá-las em questão é a de que uma cidade pertence a um estado.

Para relacionar as informações entre as tabelas, precisamos criar chaves estrangeiras, ou *foreign keys*. Ao determinarmos esse relacionamento, garantimos a integridade das informações e ainda facilitamos o processo para criar consultas mais complexas, com resultados mais detalhados. Podemos dizer que o tipo de relacionamento entre as tabelas `states` e `cities` é **1:N**, ou seja, um estado tem muitas cidades. Para o nosso exemplo, não precisamos aprofundar muito na teoria sobre relacionamentos entre tabelas, mas apenas como criá-los usando CakePHP.

Um relacionamento 1:N permite que os elementos da tabela A tenham um relacionamento com vários elementos da tabela B. Porém, os elementos da tabela B podem estar relacionados a apenas um elemento da A. As migrations permitem a criação de chaves estrangeiras usando o método `addForeignKey()`.

> Convenção para 1:N
>
> O CakePHP define uma convenção para criar referências entre as tabelas. Para referenciar uma chave de outra tabela, devemos criar uma coluna com o nome da tabela referenciada em minúsculo seguido do sufixo *id* — por exemplo, com `state_id`, informamos que existe uma ligação com a chave primária da tabela `state`.

Seguindo com nosso exemplo, se pretendemos criar uma chave estrangeira entre a tabela `cities`, devemos usar o método `addForeignKey()` na migration que criará esse relacionamento:

```
$table->addForeignKey('state_id', 'states', 'id');
```

O método recebe três parâmetros: a coluna que guardará a referência, a tabela relacionada e a coluna que será usada como referência na tabela relacionada. Ou seja, nesse exemplo, será criada uma coluna chamada `state_id` que guardará valores da coluna `id` da tabela `states`. Caso executemos a migration adicionando essa linha de código, ainda não vai funcionar e vai explodir um erro no console pelo simples fato de que o método `addForeignKey()` não cria uma coluna, mas apenas o relacionamento entre colunas previamente criadas. A forma correta de adicionar uma chave estrangeira em uma tabela é:

```
$table->addColumn('state_id', 'integer');
$table->addForeignKey('state_id', 'states', 'id');
```

É necessário criar a coluna `state_id` antes de criar a referência, lembrando que os tipos de dados devem ser equivalentes — no

caso, a coluna `id` é um inteiro; logo, a coluna `state_id` deve ser um dado do tipo inteiro. Vamos melhorar ainda mais o exemplo criando uma migration para alterar a tabela `cities` e criar uma coluna e uma chave estrangeira. Execute o comando:

```
bin/cake bake migration AddStateIdToCities state_id:integer
```

Vamos modificar o código da migration criada adicionando a chave estrangeira para a tabela `states`. O resultado será:

```
<?php
use Migrations\AbstractMigration;

class AddStateIdToCities extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('cities');
        $table->addColumn('state_id', 'integer', [
            'default' => null,
            'limit' => 11,
            'null' => false,
        ]);
        $table->addIndex('name');
        $table->addForeignKey('state_id', 'states', 'id');
        $table->update();
    }
}
```

No código, temos um exemplo completo de como criar uma chave estrangeira. Vimos que primeiro adicionamos a coluna que vai armazenar um valor e somente depois adicionamos a integridade relacional entre as tabelas `states` e `cities`. Tem um detalhe no código que ainda não foi explicado: a linha `$table->addIndex('name')`. Sempre que necessário, também podemos criar índices para melhorar a performance na execução das consultas. A criação de um índice na tabela é feita usando o método `addIndex($NomeDaColuna)`.

Utilizei a mesma migration para criar tanto a chave estrangeira como o índice.

A decisão de adicionar índice ou chave estrangeira cabe à pessoa desenvolvedora, bem como analisar e decidir se é necessário ou não ter relacionamentos no banco de dados do projeto.

3.7 Inserindo dados — Seed

Em algumas tabelas, precisamos inicializar registros, seja para criar uma base de dados para teste ou para popular algumas tabelas — por exemplo, uma que represente estados, cidades, dados fixos. Para realizar essa operação, podemos criar classes chamadas `Seeds` com o comando:

```
bin/cake bake seed NomeDaTabela
```

Quando criamos um arquivo *seed*, ele é armazenado no diretório `config/Seeds`. Para exemplificar melhor e fixar o aprendizado, vamos criar um *seed* para a tabela `states`.

```
bin/cake bake seed states
```

Com a execução do comando, será criado um arquivo `StatesSeed.php`, cujo conteúdo será:

```
<?php
use Migrations\AbstractSeed;

/**
 * States seed.
 */
class StatesSeed extends AbstractSeed
{
    /**
     * Run Method.
```

```
     *
     * Write your database seeder using this method.
     *
     * More information on writing seeds is available here:
     * http://docs.phinx.org/en/latest/seeding.html
     *
     * @return void
     */
    public function run()
    {
        $data = [];

        $table = $this->table('states');
        $table->insert($data)->save();
    }
}
```

Vale lembrar que precisamos que a tabela na qual pretendemos inserir dados já esteja criada no banco de dados, ou seja, que a migration já tenha sido criada e executada. Na classe exibida anteriormente, toda a lógica para inserir dados deve ser criada no corpo do método `run()`. Instanciamos um objeto `$table` e, com a referência para a tabela `states`, passamos os dados que queremos inserir no método `insert()`.

Você pode estar se perguntando: e agora? Como é que vou criar esses dados? Respondendo à questão, devemos criar um *array* para cada registro. Cada elemento do *array* deve ser formado com o nome da coluna e o valor. Por exemplo:

```
$data = [   ['name'=>'São Paulo'],
            ['name'=>'Ceará'],
            ['name'=>'Rio de Janeiro'],
            ['name'=>'Bahia']
        ];

        $table = $this->table('states');
        $table->insert($data)->save();
```

A chave `name` do *array* de dados representa o nome de uma coluna na tabela `states`. Quando todo o set de dados estiver completo, podemos executar os arquivos `Seeds` com o comando:

```
bin/cake migrations seed
```

Caso a intenção não seja executar todos os arquivos, mas apenas um em especial, podemos especificar o arquivo `seed` com o comando:

```
bin/cake migrations seed --seed StatesSeed
```

Os nomes dos arquivos `seed` sempre vão seguir o padrão `NomeDaTabelaSeed` e, assim como nas migrations, essa é uma funcionalidade da API Phinx.

3.8 Revertendo as alterações

Se em algum momento ou depois de aplicar alguma migration for necessário reverter as alterações feitas, podemos executar um `rollback` com o comando:

```
bin/cake migrations rollback
```

A cada ciclo de alterações, o CakePHP vai criando um histórico no banco de dados das migrações executadas. Esse registro é usado pelo comando `rollback` para desfazer sempre a última alteração executada. Até esse momento, quando executamos um comando `migrate` ou `rollback`, deixamos para o framework localizar e organizar as migrations usando a data, no formato `YYYYMMDDHHMMSS`, prefixada no nome dos arquivos.

Essas datas que compõem o nome podem ser usadas como referência quando precisamos informar uma migration específica para aplicar ou reverter alguma alteração. Para informar a versão da migration, podemos utilizar o parâmetro **-t**, por exemplo:

```
bin/cake migrations migrate -t 20161010101085
bin/cake migrations rollback -t 20161010101085
```

Vamos supor que eu quero reverter as alterações feitas por uma migration na tabela `cities`. Digamos que o nome do arquivo da migration é `20190430015419_AddStateIdToCities.php`. Podemos dar um `rollback` nela executando:

```
bin/cake migrations rollback -t 20190430015419
```

Estou especificando no comando que quero dar um `rollback` na migration criada nesse momento, `20190430015419`, informação extraída do nome do arquivo.

3.9 Banco de dados AdPET

Voltando para a aplicação AdPET, vamos concluir a estrutura do banco de dados do projeto criando o restante das tabelas. Ainda precisamos criar mais quatro tabelas: `users`, `pets`, `adoptions` e `comments`. Vamos definir as entidades e os relacionamentos que compõem todo o sistema, mas agora de uma forma mais rápida, já que explicamos o conteúdo necessário para este momento.

Uma dica legal é desenhar como será a estrutura de tabelas e relacionamentos de um banco de dados usando um diagrama ER (*Entity Relationship*). Com esse diagrama, temos uma forma de descrever as entidades envolvidas no negócio, com seus atributos, e como serão formados os relacionamentos entre si.

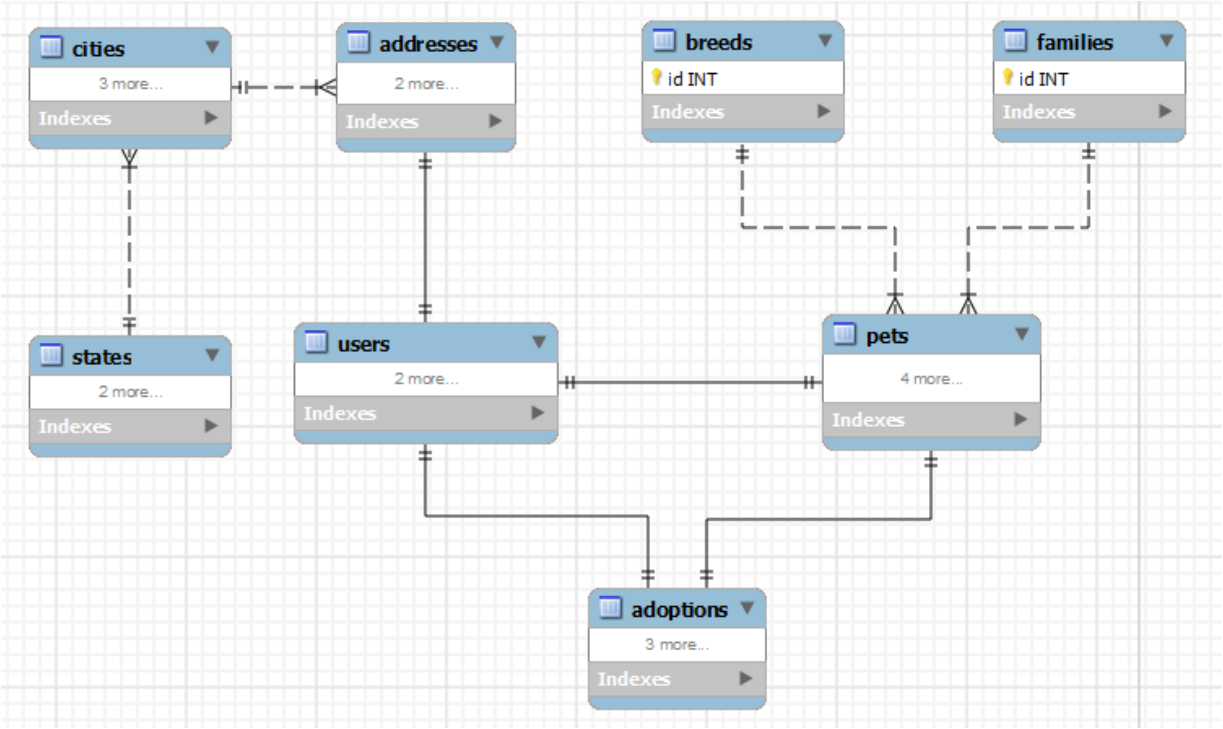


Figura 3.1: Diagrama EER.

As primeiras tabelas que construímos são relacionadas ao endereço. As tabelas `pets` e `users` vão ter um relacionamento com a tabela `addresses`, onde armazenaremos a localização do usuário e do pet. Mas ainda não criamos as migrations para elas. Vamos lá:

```
$ bin/cake bake migration createUsers
$ bin/cake bake migration createPets
```

Esses comandos não especificam de início as colunas para o `bake` criar, e isso não é nenhum problema, pois, como comentado anteriormente, podemos criar as classes das migrations com um formato mais simples. O código final será:

```
<?php
use Migrations\AbstractMigration;

class CreateUsers extends AbstractMigration
{

    public function change()
```

```
    {
        if (!$this->hasTable('users')) {
            $table = $this->table('users');

            $table->addColumn('name', 'string', ['default' => null,'limit'
=> 255,'null' => false]);
            $table->addColumn('email', 'string', ['default' =>
null,'limit' => 255,'null' => false]);
            $table->addColumn('password', 'string', ['default' =>
null,'limit' => 255,'null' => false]);
            $table->addColumn('phone', 'string', ['default' =>
null,'limit' => 255,'null' => false]);

            $table->create();
        }
    }
}
```

```
<?php

use Phinx\Migration\AbstractMigration;

class CreatePets extends AbstractMigration
{
    public function change()
    {
        if (!$this->hasTable('pets')) {
            $table = $this->table('pets');

            $table->addColumn('name', 'string', [
                'default' => null,
                'limit' => 255,
                'null' => false,
            ]);
            $table->addColumn('description', 'text', [
                'default' => null,
                'null' => false,
            ]);
            $table->addColumn('gender', 'string', [
                'default' => 'M',
```

```
                'limit' => 2,
                'null' => false,
            ]);
            $table->addColumn('birthday', 'datetime', [
                'default' => null,
                'null' => true,
            ]);
            $table->addColumn('created', 'datetime', [
                'default' => null,
                'null' => true,
            ]);
            $table->addColumn('modified', 'datetime', [
                'default' => null,
                'null' => true,
            ]);

             $table->addColumn('family_id', 'integer');
             $table->addForeignKey('family_id', 'families', 'id');

             $table->addColumn('breed_id', 'integer');
             $table->addForeignKey('breed_id', 'breeds', 'id');

             $table->addColumn('user_id', 'integer');
             $table->addForeignKey('user_id', 'users', 'id');


            $table->create();
        }
    }
}
```

Observando o código das duas migrations, vemos que, no mesmo arquivo, já criamos as colunas, relacionamentos e índices. Não é obrigatório criar primeiro as colunas e depois ir adicionando índices e chaves estrangeiras. A forma como a construção aconteceu no decorrer do capítulo teve fins didáticos para que víssemos o passo a passo.

A tabela `pets` vai representar os animais de estimação cadastrados pelos usuários do sistema e tem relacionamento com as tabelas `breeds` e `families`, que agregam informações sobre os animais e informam a qual raça e família eles pertencem, respectivamente. As duas últimas tabelas que precisam ser criadas são:

```
$ bin/cake bake migration CreateAdoptions
$ bin/cake bake migration CreateComments
```

```
<?php
use Migrations\AbstractMigration;

class CreateAdoptions extends AbstractMigration
{
    public function change()
    {

        if (!$this->hasTable('adoptions')) {
            $table = $this->table('adoptions');

            $table->addColumn('created', 'datetime', ['default' => null,
'null' => true]);
            $table->addColumn('modified', 'datetime', ['default' =>
null,'null' => true]);

            $table->addColumn('user_id', 'integer');
            $table->addForeignKey('user_id', 'users', 'id');

            $table->addColumn('pet_id', 'integer');
            $table->addForeignKey('pet_id', 'pets', 'id');

            $table->create();
        }
    }
}
```

```
<?php
use Migrations\AbstractMigration;

class CreateComments extends AbstractMigration
```

```
{

    public function change()
    {

        if (!$this->hasTable('comments')) {
            $table = $this->table('comments');

            $table->addColumn('content', 'text', [
                'default' => null,
                'null' => false,
            ]);

            $table->addColumn('created', 'datetime', ['default' => null,
'null' => true]);
            $table->addColumn('modified', 'datetime', ['default' =>
null,'null' => true]);

            $table->addColumn('user_id', 'integer');
            $table->addForeignKey('user_id', 'users', 'id');

            $table->addColumn('pet_id', 'integer');
            $table->addForeignKey('pet_id', 'pets', 'id');

            $table->create();
        }
    }
}
```

A tabela `adoptions` vai ser usada para informar o usuário que pretende adotar um animal de estimação. Nessa tabela, guardamos uma referência para a tabela `users` e outra para a tabela `pets`. Vale lembrar que as colunas `id` não precisam ser definidas. O CakePHP vai se encarregar de criá-las no momento da execução das migrations. Não esqueça também que, para aplicar as novas migrations, ainda precisamos executar o comando:

```
bin/cake migrations migrate
```

Um momento! Ainda falta um detalhe: precisamos criar as referências da tabela `addresses` com `cities` e `users`. Cada endereço dessa tabela precisa pertencer a uma cidade e a um usuário. Nesse caso, vamos criar uma migration para modificar essa tabela:

```
   bin/cake bake migration AddCityIdToAddresses city_id:integer
user_id:integer
```

```
<?php
use Migrations\AbstractMigration;

class AddCityIdToAddresses extends AbstractMigration
{

    public function change()
    {
        $table = $this->table('addresses');
        $table->addColumn('city_id', 'integer', [
            'default' => null,
            'limit' => 11,
            'null' => false,
        ]);
        $table->addForeignKey('city_id', 'cities', 'id');
         $table->addColumn('user_id', 'integer', [
            'default' => null,
            'null' => true,
        ]);
        $table->addForeignKey('user_id', 'users', 'id');
        $table->update();
    }
}
```

Pronto, agora sim! Temos a nossa estrutura no banco de dados pronta para uso.

Conclusão

Este capítulo abordou os principais conceitos envolvendo migrations, forma escolhida para criar o banco de dados do sistema AdPET. Com

a leitura deste capítulo, você foi capaz de criar tabelas, colunas e modificar estruturas já existentes. Esse passo é importante, pois é a base necessária para quando, mais adiante neste livro, formos estudar sobre *Models*, a camada Modelo da arquitetura MVC.

Capítulo 4
Criando rotas

O meio pelo qual os usuários poderão acessar páginas de uma aplicação web é usando uma URL. Para a navegabilidade da aplicação, precisamos criar rotas que acessem páginas, imagens, áudios, entre outros objetos que serão de uso do usuário. As URLs, quando bem planejadas, podem até transmitir com uma única linha informações sobre o contexto da página que estamos acessando.

Obviamente, à medida que o sistema cresce, a quantidade de rotas que precisam ser criadas também aumenta e é fundamental ter um código que facilite essa configuração e que garanta uma organização necessária para futuras manutenções.

Para dar continuidade ao nosso projeto, precisamos aprender como mapear endereços com o CakePHP. Por mais básica que seja uma aplicação web, ela terá pelo menos uma página inicial, ou seja, uma rota de acesso. Na aplicação AdPET, teremos que configurar rotas para a tela inicial, para ver detalhes dos pets, para realizar login ou para acessar cadastros, entre outras.

Qualquer definição de rota no CakePHP deve ser feita no arquivo `routes.php`, localizado em `config/routes.php`. Basicamente, a configuração de uma rota indicará qual classe *controller* vai processar a requisição vinda do usuário e qual *action* será executada. Atualmente, desde a versão 3 do CakePHP, podemos trabalhar com duas maneiras de criar uma rota:

A primeira é usando métodos estáticos, compatíveis com versões anteriores do framework. Segue um exemplo:

```
Router::connect('/', [
    'controller' => 'Pets',
    'action' => 'index']);
```

E a outra opção, mais performática e recomendada pelo CakePHP, é criar escopos para rotas:

```
Router::scope('/', function ($routes) {
   $builder->connect('/', [
        'controller' => 'Pets',
        'action' => 'index']);
});
```

Nesse segundo exemplo, quando a página inicial for requisitada (lembrando que o símbolo `/` indica a página *default*), o *controller* chamado `Pets`, que possui a *action* `index`, vai receber uma requisição e executar alguma lógica. Nos tópicos seguintes deste capítulo, vamos aumentar o nosso conhecimento sobre o framework CakePHP aprendendo as principais maneiras de criar rotas de acesso e como configurá-las.

4.1 Rotas com escopo

Criar rotas com escopos é a melhor maneira de organizar o código quando estamos trabalhando com CakePHP, pois podemos agrupar determinadas rotas por contexto. Por exemplo, podemos ter rotas para acessar páginas de uma interface administrativa ou agrupar as que vão redirecionar para os relatórios. Isso garante que o código tenha uma legibilidade bem melhor do que simplesmente escrevê-las sem contexto.

Os exemplos a seguir mostram como podemos criar vários escopos de acordo com a necessidade da aplicação:

```
Router::scope('/', function ($routes) {
    $builder->fallbacks();
});

Router::scope('/admin', function ($routes) {
    $builder->fallbacks();
```

```
});

Router::scope('/api', function ($routes) {
    $builder->fallbacks();
});
```

Veja que temos três escopos configurados: no primeiro, podemos escrever rotas que acessam recursos gerais na aplicação; no segundo, podemos agrupar rotas relativas à área administrativa; no terceiro, podemos definir rotas que fazem parte de uma API externa *RESTful*. Dentro desses escopos, podemos adicionar quantos endereços a aplicação precisar, é uma forma mais legível e eficiente de codificar nossas rotas de acesso.

O método `$builder->fallbacks();` é uma maneira simplificada de usar *dash* na escrita da configuração de rota. A outra forma que temos para usar o `DashedRoute` é exemplificada a seguir:

```
$routes->connect('/:controller/:action/:id',
    ['action' => 'show'],
    ['routeClass' => 'DashedRoute']);

$routes->connect('/:controller',
    ['action' => 'delete'],
    ['routeClass' => 'DashedRoute']);
```

Dessa maneira, precisamos passar o `routeClass` em todas as rotas configuradas. Usando o `fallbacks`, definimos um comportamento padrão para qualquer rota criada dentro do *scope*. O `routeClass` não é um atributo obrigatório, mas, caso não seja usado, podemos gerar URLs inconsistentes ou mal formatadas.

Quando passamos a classe `DashedRoute`, estamos garantindo que os endereços serão convertidos para letras minúsculas e *dashed*, ou seja, com o caractere `-` separando as palavras, por exemplo, `/meu-primeiro/endereco/ola-mundo`. Este é um formato que facilita a indexação pelos sistemas de buscas, uma técnica utilizada no SEO.

URL AMIGÁVEL — SEO

Quando estamos navegando, tentando encontrar um determinado site, um dos requisitos (entre outros) que contribui para que o site seja encontrado pelos buscadores é o de adicionar termos mais simples aos endereços e que representem, de alguma forma, o conteúdo que estamos procurando. Chamamos essa técnica de *Friendly Routing*, termo em inglês que, grosso modo, podemos traduzir como "rotas amigáveis". Portanto, SEOs (*Search Engine Optimization*) são técnicas utilizadas que, quando bem aplicadas, garantem que um site seja encontrado mais facilmente pelos sistemas de busca e fique bem posicionado nos resultados orgânicos, por exemplo, o ranking no site de busca do Google.

O CakePHP permite criar rotas amigáveis. Quando definimos um *route scope*, ao adicionar uma nova rota, podemos redefinir o nome do endereço da URL:

```
$builder->connect('nome_da_URL',
   ['nome_do_controlador' => 'metodo'],
   ['option' => 'matchingRegex']);
```

Nesse exemplo, o método `connect()` do objeto `$routes` pode receber três parâmetros, dois obrigatórios e um opcional. O primeiro parâmetro é uma *string* com o endereço da URL; o segundo é um *array* contendo o nome do controlador e o método que vai ser executado, e o terceiro parâmetro é um *array* com opções adicionais para a rota.

O segundo parâmetro ainda possibilita passar um valor que será usado para compor a rota. Imagine, por exemplo, que seja preciso criar uma rota que acesse os produtos de uma determinada categoria. Na criação dessa rota, no segundo parâmetro do método `connect()`, podemos adicionar o `id` da categoria cujos produtos

pretendemos visualizar. A seguir, um exemplo de como criar uma rota com o *id* de um objeto:

```
Router::scope('/', function ($routes) {

    $builder->connect('/eletronicos',
        ['controller' => 'Products',
         'action' => 'category', 10]);
});
```

Como passamos o `/eletronicos`, esse código geraria uma URL acessível no endereço `http://www.meusite.com.br/eletronicos`, em vez de em `http://www.meusite.com.br/products/category/10`, que seria o padrão se fosse omitido o nome para essa rota. Esse procedimento simplifica a chamada da URL, pois estamos passando o valor 10, que corresponde à categoria de eletrônicos, e toda a lógica ficará implementada no *controller* `products` e no método `category`. Como vimos anteriormente, nos conceitos sobre SEO, uma boa prática é criar nomes de endereços que indiquem qual recurso estamos tentando acessar, por exemplo:

```
Router::scope('/', function ($builder) {

    $builder->connect('/contato',
        ['controller' => 'Contacts','action' => 'show']);

    $builder->connect('/produtos/games',
            ['controller' => 'Products', 'action' => 'games']);

    $builder->connect('/noticia/governo/:id',
           ['controller' => 'Government', 'action' => 'show']);

    $builder->fallbacks('DashedRoute');
});
```

Essas configurações de rotas resultarão nos seguintes endereços:

- `http://www.meusite.com.br/contato/`
- `http://www.meusite.com.br/produtos/games/`

- `http://www.meusite.com.br/noticias/governo/2/`

Os endereços também aceitam o uso de *wildcard* `*` (ou curinga, em português) quando pretendemos passar algum valor adicional junto na URL. Podemos ter rotas com uma estrela, que informa que a rota pode ser acessada passando um parâmetro, ou adicionar duas estrelas, `/**` . Nesse caso, todo o restante da URL será interpretado com um único parâmetro. A seguir, citamos alguns exemplos de como usar curingas nas URLs — lembrando que os valores podem ser acessados no método definido pelo parâmetro `action` no *controller*.

```
Router::scope('/', function ($builder) {

    $builder->connect('/contato/*',
        ['controller' => 'Contacts', 'action' => 'show']);

    $builder->connect('/produtos/**',
        ['controller' => 'Products', 'action' => 'games']);

    $builder->fallbacks('DashedRoute');
});
```

4.2 Validando elementos das URLs

O CakePHP suporta criar uma URL com vários elementos, cujos valores serão lidos como parâmetros na classe *controller* especificada no endereço. Um elemento em uma rota é uma palavra-chave acompanhada de dois-pontos, `:` . Por exemplo, podemos criar um endereço para visualizar notícias de uma determinada data, que recebe como parâmetros o ano, mês e o dia, consecutivamente.

```
    $builder->connect(
        '/:controller/:ano/:mes/:dia',
```

```
        ['action' => 'index']
    );
```

Nesse código, a rota tem quatro elementos mapeados. O primeiro, `:controller`, é uma palavra reservada para o CakePHP cujo valor representa o nome do controlador que vai receber a requisição. No código da rota, não estamos especificando qual o *controller*; é uma forma mais genérica para criar uma rota. O controlador que receberá a requisição vai depender do valor informado no momento em que a URL for acessada. Os elementos `:ano`, `:mes` e `:dia` são parâmetros adicionais definidos pelo desenvolvedor que serão acessados no método `index()`.

Essa configuração pode gerar uma URL, por exemplo, que simule uma chamada para esta rota: `/noticias/2016/03/25`. Nela, os valores `noticia`, `2016`, `03` e `25` representam os parâmetros `:controller`, `:ano`, `:mes` e `:dia`, respectivamente. Essa rota vai funcionar corretamente sempre que os valores que representam uma data forem passados para o controlador `noticia`.

Agora, imagine que, para este mesmo exemplo, quando um usuário utilizar o sistema e escrever algum texto no parâmetro que representa o ano (que é um valor numérico), como `/noticias/test/03/25`, será que o código funcionará corretamente?

Provavelmente esse parâmetro vai gerar alguns problemas na execução, caso o código não esteja preparado para tratar exceções. Podemos evitar determinadas quebras no código validando se os parâmetros da URL estão fora do formato.

Para validar os valores de cada parâmetro individualmente, podemos utilizar expressões regulares adicionando uma expressão regular ao terceiro parâmetro do método `connect()`. Usando o exemplo anterior, teremos como validar os valores para ano, mês e dia usando o seguinte código:

```
$builder->connect(
    '/:controller/:ano/:mes/:dia',
```

```
    ['action' => 'index'],
    [
        'ano' => '[12][0-9]{3}',
        'mes' => '0[1-9]|1[012]',
        'dia' => '0[1-9]|[12][0-9]|3[01]'
    ]
);
```

Com essas expressões regulares, estamos validando e assegurando que os parâmetros recebam valores numéricos. A validação vai acontecer antes do acesso à *action* para processar a requisição. Uma observação é que, como esse mesmo exemplo, além do nome do *controller*, podemos também parametrizar a *action* que vai tratar a requisição informando o *controller* e deixando o método em aberto usando a chave `:action`. Por exemplo, poderíamos escrever:

```
$builder->connect(
    '/noticia/:action/,
    ['controller' => 'News']);
```

No código, a *action* vai depender do valor que for informado para a URL no momento do acesso. Esse tipo de configuração deve ser bem planejado pela pessoa programadora, a fim de criar uma situação em que um parâmetro lido na URL gere para o usuário um aviso de página não encontrada ou de outro problema.

> **O QUE SÃO EXPRESSÕES REGULARES?**
>
> Uma expressão regular é um padrão de caracteres que pode ser identificado em um texto. São usadas normalmente para validar palavras, endereços ou qualquer sequência de caracteres. Operações com texto, como remover ou adicionar partes de palavras, também podem utilizar expressões regulares, principalmente pela ótima velocidade computacional com que as expressões são executadas.

Endereços com prefixos

Em certas ocasiões, precisamos que as URLs disponíveis para o usuário do sistema contenham informações do recurso acessado — por exemplo, quando queremos que as URLs que fazem parte de uma API chamada *RESTful* contenham o prefixo `api`, ou que as URLs que acessam a interface administrativa comecem com `admin`. Ou seja, precisamos adicionar prefixos nos endereços citados. Com o CakePHP, podemos criar prefixos através do método estático `prefix` da classe `Router`. Segue um exemplo:

```
Router::prefix('admin', function ($builder) {

    $builder->connect('usuarios/visualizar/:id',
        ['controller' =>'users', action' => 'view']);

    $builder->connect('usuarios/editar/:id',
        ['controller' =>'users', action' => 'edit']);

    $builder->fallbacks('DashedRoute');
});
```

Nesse exemplo, o endereço gerado seria `/admin/usuarios/editar/10` e `/admin/visualizar/editar/10`. Todas as rotas acionadas abaixo do prefixo *admin* sempre serão iniciadas com a palavra *admin* (lembrando que o número 10 mencionado no exemplo é o valor atribuído ao elemento `:id`, para ilustrar melhor).

Passando valores

Uma rota pode conter informações que serão utilizadas na lógica da *action* especificada. O CakePHP permite declarar um método no *controller* cujos parâmetros correspondem aos elementos passados na rota. Por exemplo:

```
$routes->connect('artigo/:id/:ano', [
    'controller' => 'article' ,
    'action' => 'view',
     ['pass' => ['id','ano']
]);
```

Nesse trecho de código, a opção `pass` vai informar ao framework para permitir acessar os valores dos parâmetros `:id` e `:ano` no método `view()`. No *controller*, o método `view()` deve ser escrito recebendo esses dois parâmetros.

```
//src/Controller/ArticlesController.php
public function view($id = null, $ano = null)
{
    //adicionar lógica
}
```

Essa é uma boa maneira para criar nossas *actions* nas classes *controllers*, embora não seja a única maneira para acessar parâmetros. Podemos também usar o objeto `$request` para acessá-los. No entanto, saber quais parâmetros já estamos esperando na *action* melhora a legibilidade e o entendimento de qual deverá ser o comportamento em questão. Mais detalhes sobre o objeto `$request` serão tratados no capítulo 6, sobre *controllers*.

4.3 Criando rotas AdPET

Agora temos o conhecimento necessário para criar os endereços da aplicação AdPET. Começando pela página inicial do sistema, a aplicação vai exibir uma lista de pets cadastrados pelos usuários e, caso um item dessa lista seja clicado, uma página com mais detalhes sobre o animal será exibida. Este é um pequeno caso de uso do site que vamos utilizar para iniciar o desenvolvimento das rotas.

Editando o arquivo `routes.php` no diretório `config`, vamos criar duas rotas que serão interceptadas pelo controlador `PetsController`. Para isso, devemos adicionar o código:

```
$routes->setRouteClass(DashedRoute::class);

$routes->scope('/', function (RouteBuilder $builder) {
```

```
    $builder->connect('/',
        ['controller' => 'Pets', 'action' => 'index']);


    $builder->connect('/pet/:id',
        ['controller' => 'Pets', 'action' => 'view'],
        [
          'pass' => ['id'],
           'id' => '[0-9]+'
        ]);

    $builder->fallbacks();
});
```

No momento, só precisamos adicionar essas duas rotas, que serão alvos de estudo no capítulo sobre *controllers*. No decorrer dos estudos, mais rotas serão adicionadas ao arquivo. Nesse código, temos duas rotas que acessam o *controller* `PetsController`, porém em métodos diferentes. O método `index()` será responsável por exibir uma lista de pets na página inicial e o método `view()` vai exibir detalhes de um pet selecionado.

Uma observação sobre a rota `/pet/:id` é que estamos adicionando o elemento `:id`, que está sendo validado por uma expressão regular `[0-9]+`, cuja função é permitir apenas valores numéricos. A opção `pass` permite que o método `view()` acesse o valor `id` no escopo da função, exemplo da *action* a seguir:

```
//src/Controller/PetsController.php
public function view($id = null)
{
    //adicionar lógica
}
```

Pronto! Feito isso, o próximo passo é entender como trabalhamos com o *controller*. Por isso, a implementação da classe `PetsController` será detalhada mais adiante.

Conclusão

Neste capítulo, aprendemos como criar rotas no CakePHP, principalmente usando o conceito de criação de rotas para um determinado escopo. Vimos em qual arquivo devemos adicionar e configurar tudo relacionado ao roteamento. Vimos ainda como passar parâmetros na URL e tomamos conhecimento de que, via de regra, toda requisição vai ser tratada em uma *action* de um *controller* especificado.

Capítulo 5
Middlewares

Desde que foi lançada a versão 3.x, o CakePHP incorporou em sua estrutura o uso de *middlewares*, que podem ser definidos como uma camada oculta da aplicação na qual dados trafegam de um lado para o outro antes de chegar à aplicação. No CakePHP, os *middlewares* encapsulam a aplicação de forma que as requisições e as respostas, antes de chegarem ao destino final, trafeguem por essas camadas. Para aplicações PHP, o padrão a ser seguido para a implementação de *middlewares* é o *PSR-7 Request & Response Interface*, que descreve como serão as interfaces para representar requisições HTTP.

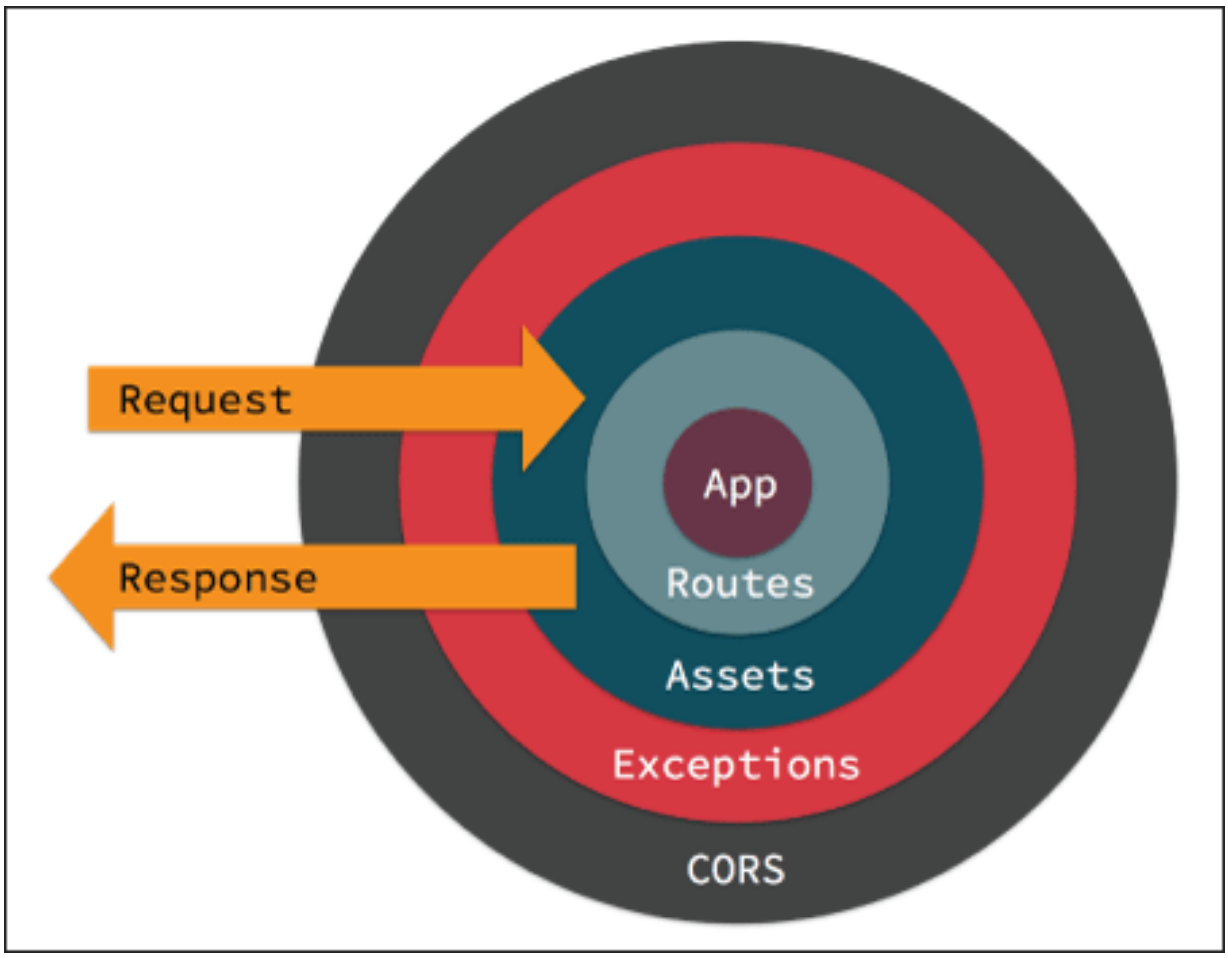


Figura 5.1: Exemplo das camadas do middleware envolvendo a aplicação. Fonte: CakePHP, c2022.

Você pode estar se perguntando aqui: por que conhecer sobre *middlewares* é importante?

Bem, posso afirmar que desde que estudamos sobre rotas no capítulo anterior já estávamos usando um *middleware*, o `Cake\Routing\Middleware\RoutingMiddleware`. Por padrão, o CakePHP possui alguns já definidos como:

- `Cake\Routing\Middleware\RoutingMiddleware`.
- `Cake\Http\Middleware\CsrfProtectionMiddleware`.
- `Cake\Http\Middleware\BodyParserMiddleware`.
- `Cake\I18n\Middleware\LocaleSelectorMiddleware`.
- `Cake\Http\Middleware\SecurityHeadersMiddleware`.

- `Cake\Error\Middleware\ErrorHandlerMiddleware` .
- `Cake\Routing\AssetMiddleware` .
- `Cake\Http\Middleware\EncryptedCookieMiddleware` .

As requisições HTTP, por exemplo, acessadas pelas rotas, são processadas pela classe `Cake\Routing\Middleware\RoutingMiddleware` . Qualquer requisição que contenha informação que vem pelo protocolo HTTP utiliza essa classe como mediadora de controle das informações que chegam e saem da aplicação.

O uso de *middleware* é um padrão bem utilizado nos mais diversos *frameworks*. A autenticação, por exemplo, é ótima para entender sua ação no sistema. Nela, o *middleware* será o responsável por verificar as credenciais presentes na requisição e permitir que a requisição chegue a um *controller* caso passe pela lógica que a valida ou a invalida. Mais adiante, no capítulo sobre autenticação, vamos ver como configurar isso no projeto AdPET.

5.1 Criando middlewares

Como visto, o CakePHP vem com um conjunto de *middlewares* que fornece funcionalidades para diversas áreas da aplicação. No entanto, o desenvolvimento de software nunca é linear e sempre vamos precisar criar algo para suprir uma necessidade bem específica do projeto. Com o CakePHP, podemos criar nossos próprios *middlewares* para lidar com *Cors*, *Upload*, *Cookie* ou qualquer outra situação, desde que seja oriunda de uma requisição HTTP. Todos os *middlewares* criados pelo desenvolvedor têm que estar no diretório `src/Middleware` , seguindo a convenção do framework.

Para exemplificar, criamos um *middleware* adaptado da documentação oficial do CakePHP para criar um cookie para a

aplicação AdPET. O nome do arquivo será `AdPETCookieMiddleware`. Vamos ao código:

```
<?php

namespace App\Middleware;

use Cake\Http\Cookie\Cookie;
use Cake\I18n\Time;

class AdPETCookieMiddleware
{
    public function __invoke($request, $response, $next)
    {

        $cookieName = 'adpet';
        $cookie = $request->getCookie($cookieName);

        if (!$cookie) {
            $expiry = new Time('+ 1 year');
            $response = $response->withCookie(new Cookie(
                $cookieName,
                $request->getRequestTarget(),
                $expiry
            ));
        }

        return $next($request, $response);
    }
}
```

Observando o código, podemos destacar que, para escrever um *middleware* no CakePHP, temos que obedecer às seguintes regras:

- Retornar um objeto que implemente o `PSR-7 ResponseInterface`;
- Sobrescrever o método `__invoke`;
- O nome da classe e do arquivo têm que ter o sufixo `Middleware`.

É dentro do método `__invoke` que toda a mágica acontece. Temos acesso às informações que vêm junto do `request` do usuário e temos como modificar a resposta. Isso deixa bem exemplificado e de forma didática qual é a principal função e o que se pode fazer com um *middleware*.

5.2 E aí, como usar?

Muito foi dito, mas, e aí, como vou usar *middlewares*? Isso é o mais simples de tudo, agora que sabemos a principal intenção para usá-los. Todos os *middlewares* são carregados na aplicação pela classe `Application`, especificamente no `src/Application.php`. Nessa classe, precisamos declarar tudo o que for necessário no método `middleware`:

```
namespace App;

use Cake\Http\BaseApplication;
use Cake\Error\Middleware\ErrorHandlerMiddleware;

class Application extends BaseApplication
{
    public function middleware($middlewareQueue)
    {
        ...
        $middlewareQueue->add(new AdPETCookieMiddleware());
        return $middlewareQueue;
    }
}
```

A variável `$middlewareQueue` é um objeto da classe `MiddlewareQueue`, responsável por adicionar na aplicação *n* instâncias dos *middlewares*. Ou seja, com o CakePHP, para cada novo *middleware* necessário, para que ele funcione na aplicação, seja ele plugin de terceiros ou

feito por você, precisamos declará-lo na classe `Application`, como demonstramos no código.

Conclusão

É fundamental essa abordagem sobre os middlewares, porque esse conhecimento vai facilitar o entendimento de como rotas e controles estão conectados, como de fato acontece quando as informações que trafegam com o protocolo HTTP são acessadas pelo CakePHP. No próximo capítulo, aprenderemos o que são *controllers* e como usá-los.

Referências

CAKEPHP. (Site institucional). *Middleware*. c2022. Disponível em: https://book.cakephp.org/4/en/controllers/middleware.html. Acesso em 16 nov. 2022.

Capítulo 6
Controllers

O CakePHP segue o *design pattern* **MVC** (Model-View-Controller, ou modelo-visão-controlador, em português), um padrão arquitetural que divide a aplicação em camadas. Distribuir a lógica da aplicação em camadas organiza o código desenvolvido, facilita a manutenção e permite que desenvolvedores de software trabalhem em conjunto, implementando partes sem afetar o trabalho do outro; assim, funcionalidades são escritas com mais produtividade.

Cada camada no padrão MVC tem uma função específica:

- **Model**: representa os dados da aplicação;
- **View**: apresenta as informações da aplicação, por exemplo, as páginas;
- **Controller**: trata as requisições em conjunto com as camadas View e Model.

O *controller*, alvo deste capítulo, tem uma associação direta com as rotas configuradas. Quando configuramos uma rota no arquivo `routes.php`, podemos especificar qual o *controller* e a *action* que vai tratar a requisição; ou seja, para dar continuidade ao fluxo da requisição, vamos precisar criar um *controller* para responder às requisições.

Por exemplo, para exibir uma lista de pets, precisamos do seguinte trecho de código no arquivo `routes.php`.

```
Router::scope('/', function ($builder) {

    $builder->connect('/pets',
        ['controller' => 'Pets', 'action' => 'index']);

});
```

Nesse código, estamos criando uma rota, `http://localhost:8765/pets`. Quando acessada pelo usuário, ela gera uma requisição que vai ser tratada no *controller* `PetsController` e na *action* `index()` e, então, processará uma resposta para o usuário.

O CakePHP utiliza as classes `Cake\Routing\Router` e `Cake\Routing\Dispatcher` para localizar e criar uma instância do controlador especificado na rota e passar um objeto `Cake\Http\ServerRequest`, que encapsula os dados pertinentes da requisição para a *action*.

O fluxo completo de uma requisição HTTP no CakePHP, passando por todas as camadas, é bem representado pela imagem a seguir:

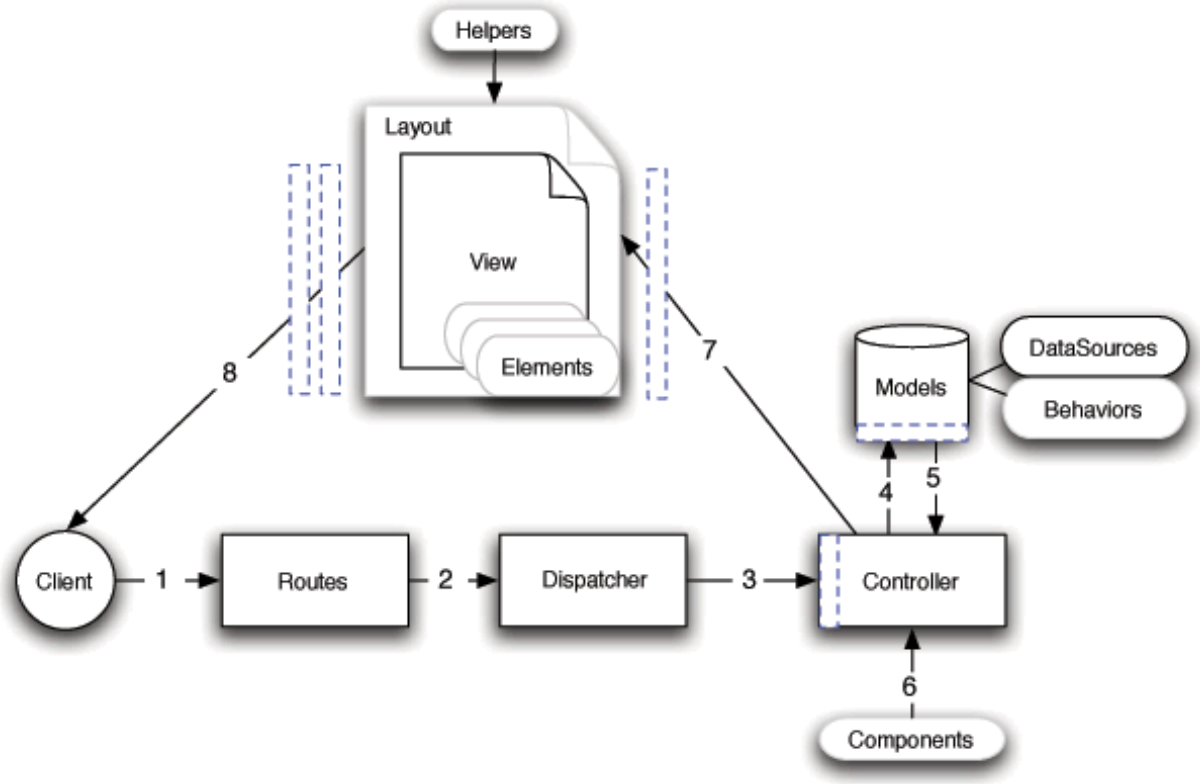


Figura 6.1: Fluxo de uma requisição. Fonte: http://book.cakephp.org.

Analisando o fluxo da requisição, que é iniciado quando o usuário interage e acessa uma rota da aplicação, a rota delega para o controlador a requisição vinda do usuário, para então processar uma

resposta para o usuário em conjunto com o modelo, que, por sua vez, acessa o banco de dados caso seja necessário. Quando a resposta está concluída, ela é passada para uma *view* processar a informação e apresentá-la de forma adequada para o usuário.

Basicamente, o *controller* tem meios para acessar a camada de visão e a camada de modelo, ou seja, o *controller*, além de receber requisições e processar uma resposta, passa informação para as *views* e interage com os modelos, que são classes que representam as tabelas do banco de dados.

6.1 Criando um controller

Todos os *controllers*, por convenção do framework, devem ser criados no diretório `src/Controller`. Podemos criá-los manualmente ou usando o comando do `bake` para criar o arquivo:

```
cake bake controller NomeDoArquivo
```

Ao executar esse comando, passando o nome do novo *controller*, será criado um arquivo dentro do diretório `src/Controller` contendo cinco *actions*: `index()`, `view()`, `add()`, `edit()` e `delete()`. Elas definem as operações básicas para trabalhar com dados em um sistema que utiliza um banco de dados relacional.

É notado que esse comando gera uma porção de código extra com o intuito de acelerar o desenvolvimento, fornecendo uma estrutura que, certamente, em algum momento, o desenvolvedor vai precisar criar. Em outras palavras, esses métodos do *controller* podem ser chamados de **CRUD**, acrônimo para *Create(C), Retrieve(R), Update(U)* e *Delete(D)* (ou, em português, Criar, Consultar, Atualizar e Deletar, respectivamente.

NOMES DOS *CONTROLLERS*

Como visto no início do livro, a convenção adotada pelo CakePHP que padroniza a nomenclatura para os *controllers* é a de que todos os nomes devem estar no plural, seguidos do prefixo `Controller` e usando o padrão CamelCase — por exemplo, `PetsController`, `BrandController`, `MyNamesController` são nomes válidos para *controllers*. Uma recomendação é também deixar os nomes no idioma inglês. Isso vai evitar problemas com outras convenções que o CakePHP utiliza.

A estrutura básica de um *controller* é:

```
<?php
namespace App\Controller;

use App\Controller\AppController;

class PetsController extends AppController
{
    //criar actions aqui
}
```

Para criar comportamentos comuns entre os *controllers* criados na aplicação, precisamos fazer com que a classe `PetsController` estenda a classe `AppController`, localizada em `src/Controller/AppController.php`. Como visto anteriormente, em `AppController`, podemos criar métodos e propriedades que podemos reusar em outros controladores. É uma estrutura simples sugerida pelo CakePHP para ajudar a evitar código duplicado no projeto. O esqueleto da classe `AppController` pode ser visualizado a seguir:

```
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
```

```
{
  // seus métodos aqui
}
```

No capítulo 4, sobre rotas, na continuação do projeto AdPET, declaramos duas rotas que apontam para o `PetsController`, uma para listar os pets e outra para pegar informação de um único pet.

```
Router::scope('/', function (RouteBuilder $builder) {


    $builder->connect('/',
        ['controller' => 'Pets', 'action' => 'index']);


    $builder->connect('/pet/:id',
        ['controller' => 'Pets', 'action' => 'view'],
        [
          'pass' => ['id'],
           'id' => '[0-9]+'
        ]);

    $builder->fallbacks();
});
```

Pegando o gancho dessa seção, vamos praticar criando o `PetsController` com todos os métodos CRUD, o que será necessário para o funcionamento dessas duas rotas da aplicação. Para isso, vamos executar o seguinte comando:

```
cake bake controller Pets
```

O *controller* `PetsController`, criado com esse comando mostrado anteriormente, terá duas *actions*, `index()` e `view()`, como requerido na declaração das rotas `/` e `/pet/id`. Com essa estrutura de código criada, já é possível fazer requisições para esses endereços das rotas, pois temos um *controller* que vai tratar as requisições e

fornecer a resposta adequada. Esses dois métodos têm a seguinte estrutura:

```
public function index()
{
     $pets = $this->paginate($this->Pets);

     $this->set(compact('pets'));
}

public function view($id = null)
{
    $pet = $this->Pets->get($id, [
        'contain' => []
    ]);

    $this->set('pet', $pet);
}
```

Essas duas *actions*, `index()` e `view()`, quando requisitadas, têm a função, respectivamente, de listar todos os pets e visualizar com mais detalhes as informações de cada um. Mais adiante, no capítulo 10 (sobre *view*), vamos mudar um pouco essa implementação e veremos como exibir as informações tratadas nos métodos.

Objeto Request e Response

Quando uma *action* é acessada no *controller*, é criado um objeto `Cake\Http\ServerRequest`, disponibilizado pela variável `$this->request`. Esse é o objeto que o CakePHP utiliza para representar uma requisição HTTP que vai fazer a função das variáveis `GET`, `POST`, `FILES` do PHP; ou seja, com o objeto `this->request`, podemos acessar os parâmetros enviados pelos formulários, URLs, além de fornecer informações sobre domínio, IP e cabeçalhos.

No capítulo anterior, quando estudamos sobre middlewares, vimos que o CakePHP tem o `Cake\Routing\Middleware\RoutingMiddleware`. Esse middleware vai interceptar as requisições das rotas e repassar a

requisição adiante até chegar ao *controller*; portanto, o objeto `this->request` implementa o padrão *PSR-7 ServerRequestInterface*. Caso você queira saber mais sobre esse padrão, acesse: https://www.php-fig.org/psr/psr-7/.

6.2 Acesso a dados

Acessando dados via GET

É bem comum passar valores junto com uma requisição GET, escrevendo parâmetros na URL que acessam algum recurso do sistema. Por exemplo, quando precisamos paginar os resultados de uma consulta, são passados parâmetros que representam a página ou algum valor que seja usado na pesquisa dos dados. Esses parâmetros são conhecidos como *query string*. Com o CakePHP, podemos recuperar esses valores usando o método `Cake\Http\ServerRequest::getQuery($name)`.

Na URL `http://wwww.exemplo.com?page=2&query=test`, é passado o parâmetro `page` com o valor `2` e o parâmetro `query` com o valor `test`. Para ler esses valores, fazemos:

```
$this->request->getQuery('page');
$this->request->getQuery('query');
```

Fácil, não é? Caso um parâmetro acessado pelo método `$this->request->getQuery()` não exista na URL, será retornado o valor `NULL`. Sabendo disso, podemos sempre verificar os dados para evitar futuros erros no código, como no exemplo a seguir:

```
if($this->request->getQuery('page') != NULL){
    $this->request->getQuery('page');
}
```

Para acessar todos os parâmetros enviados, utilizamos o método `Cake\Http\ServerRequest:: getQueryParams`.

```
$query = $this->request->getQueryParams();
```

Acessando dados via POST

Quando o verbo da requisição HTTP é um POST, não é possível acessar os parâmetros passados pelo método `Cake\Http\ServerRequest::getQuery($name)`, mas, sim, pelo método `Cake\Http\ServerRequest:: getData($name)`. Parâmetros passados via POST não são expostos como no GET. Eles não são visualizados na URL; por isso, o método POST é utilizado em formulários para registro de dados e autenticação de usuários, por exemplo.

Citando exemplos de código para ler dados via POST de um formulário, podemos escrever o seguinte código:

```
$this->request->getData('email');
$this->request->getData('username');
$this->request->getData('password');
```

Nos exemplos anteriores, o código retornará os valores passados nos parâmetros `email`, `username` e `password`, respectivamente. Em algumas ocasiões, usamos *array* para enviar informações — nesse caso, acessamos *data* como um *array* de dados. O código a seguir exemplifica isso:

```
$this->request->data['nome'];
```

Ainda é possível ler dados de estruturas maiores que usam *array* com mais de uma dimensão. Veja o código seguinte:

```
$this->request->data['comments'][1]['author'];
$this->request->data['comments'][1]['message'];
```

Nesse exemplo, `comments` é um *array* e os elementos dentro deste *array* são outros *arrays* de dados, porém podemos facilmente ler os valores com o método **data**. Você deve ter percebido que, quando o verbo HTTP muda, o método que vai tratar a requisição também muda. Basicamente, até agora, vimos que, para ler dados via GET,

usamos o método `getQuery()`; para os dados enviados com o POST, é preciso usar o método `getData()`.

Acessando dados via PUT, PATCH e DELETE

Agora você deve estar se perguntando qual será o novo método. Bem, para nossa conveniência, quando realizamos requisições usando DELETE ou PUT que contenham dados no corpo da requisição, podemos utilizar o mesmo método `Cake\Http\ServerRequest::getData($name)` para acessar o *request body data*.

6.3 Checando as requisições

Em determinado momento, precisamos testar em nossas *actions* que tipo de requisição estamos recebendo para evitar algum problema na execução do código ou para passar para o usuário uma mensagem de erro mais adequada — recurso muito útil quando estamos criando uma API RESTful.

O CakePHP disponibiliza o método `Cake\Http\ServerRequest::is($type)` para realizar verificações. Por exemplo, para verificar qual é o verbo de uma requisição HTTP, podemos utilizar as seguintes linhas:

- `is('post')` : verifica se a requisição é um POST;
- `is('get')` : verifica se a requisição é um GET;
- `is('put')` : verifica se a requisição é um PUT;
- `is('delete')` : verifica se a requisição é um DELETE;
- `is('patch')` : verifica se a requisição é um PATCH.

```
if($this->request->is('post')){
    //executar lógica
```

```
}
```

Existem ainda outras informações que podem ser verificadas com o método `is()`:

- `is('head')`: verifica se a requisição é um HEAD;
- `is('options')`: verifica se a requisição é um OPTIONS;
- `is('ajax')`: verifica se a requisição é Ajax, ou seja, uma `XMLHttpRequest`;
- `is('ssl')`: verifica-se se a requisição foi com SSL;
- `is('flash')`: verifica se o *User-Agent* é do tipo *Flash*;
- `is('requested')`: verifica se a requisição tem um parâmetro chamado `requested`;
- `is('json')`: verifica se o tipo da requisição é `application/json`;
- `is('xml')`: verifica se o tipo da requisição é `application/xml`

6.4 Uploads de arquivos

Nos sistemas de informação, é bem comum a necessidade de enviar arquivos, seja para enviar uma foto do usuário, arquivos para serem processados etc. Toda vez que enviamos um arquivo para o servidor, estamos executando um *upload*. Podemos usar o CakePHP para acessar os arquivos enviados também pela classe `Cake\Http\ServerRequest` com o método:

```
$file = $this->request->getData('file');
```

Ou com o método:

```
$file = $this->request->getUploadedFile('file');
```

Independente de qual for o método usado para ter acesso ao arquivo enviado, o retorno pode ser `NULL` quando nenhum arquivo for encontrado na requisição ou quando for retornado um objeto que implementa a interface `\Psr\Http\Message\UploadedFileInterface`. Para

representar o arquivo, o CakePHP disponibiliza os seguintes métodos:

```
$nome = $file->getClientFilename();
$tipo = $file->getClientMediaType();
$tamanho = $file->getSize();
$nomeTemporario = $file->getStream()->getMetadata('uri');
$error = $file->getError();
```

Com as informações do arquivo, podemos trabalhar de forma bem fácil, ganhar tempo e aumentar a produtividade. Ainda podemos mover o arquivo para o diretório desejado usando `moveTo()`. Por exemplo:

```
$file->moveTo($novoCaminhoDeArquivo);
```

Uma variação do método `$this->request->getUploadedFile('$name')` é o `$this->request->getUploadedFiles()`, que retorna um *array* de objetos representando os arquivos. É óbvio que usaremos este método se o formulário enviar a lista de arquivos para o servidor. Um exemplo prático da utilização desse método será abordado no capítulo final do cadastro de pets, no qual enviamos para o servidor uma foto do animal para compor o seu perfil.

6.5 Respostas HTTP

O CakePHP trata as respostas HTTP usando a classe `Cake\Http\Response` e uma instância desta pode ser acessada pela variável `$this->response` nos *controllers*. A classe `Response` tem métodos que nos permitem mudar informações no *header* das respostas; por exemplo, ela é muito útil quando é preciso fazer um download de um arquivo, definir *Character Set*, *Content Types*, *Cache* do browser, entre outras. No decorrer do livro, serão apresentadas mais funções sobre a class `Response` em um contexto

mais didático, porém, a seguir, já temos alguns exemplos dos métodos disponíveis:

```
public function index(){

    $this->response->withType('UTF-8');
    $this->response->withDisabledCache();
    $this->response->withHeader('Location', 'http://example.com');
    $this->response->withStringBody($messages);
}
```

Nesses exemplos, estamos alterando o `response` da *action* `index()`, passando o UTF-8 para o *charset*, que é o termo usado para denominar a codificação de caracteres. Estamos também desabilitando a cache do *browser* e modificando o *header*, escrevendo uma mensagem no corpo da resposta. Lógico, isso não faz muito sentido usado dessa maneira, é apenas para exibir alguns métodos.

6.6 Finalizando a classe PetsController

Precisamos fazer algum treino com o que foi aprendido até agora para fixar melhor o conteúdo. Agora temos a possiblidade de criar uma estrutura para receber uma requisição e devolver uma resposta, pois já abordamos rotas, *controllers* e objetos de `Request` e `Response`. Então, voltando para o projeto AdPET, foram criadas duas rotas no arquivo `route.php` que apontam para a classe `PetsController`.

Primeiro, precisamos subir nosso servidor HTTP usando o comando `bin\cake server`. Quando tentarmos visualizar a aplicação no endereço `http://localhost:8765/`, ocorrerá o seguinte problema:

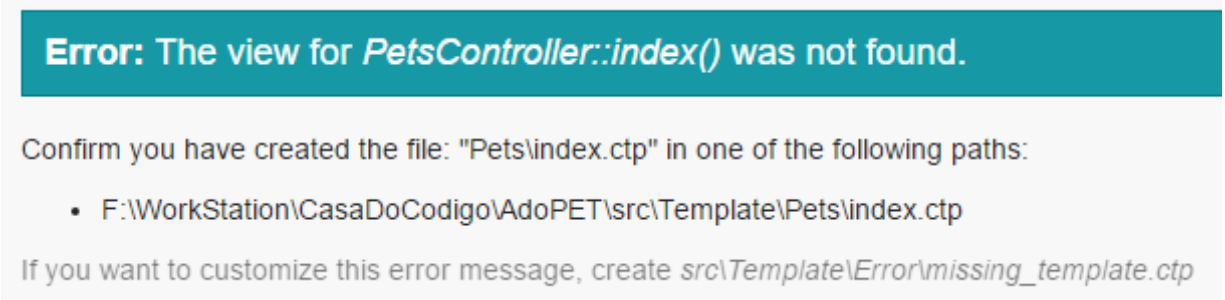


Figura 6.2: Error: The view for PetsController::index() was not found.

Isso acontece porque, quando acessamos uma *action* em um *controller*, por padrão, o CakePHP responde renderizando uma página HTML para o usuário. Nesse exemplo, para não ocorrer novamente esse problema, basta apenas criar um arquivo chamado `index.php` no diretório `template/Pets/` — maiores detalhes sobre *views* serão abordados no capítulo 10, dedicado apenas à visualização de informações.

Para exibir uma mensagem diferente que não seja um erro, podemos usar o objeto `response` para reescrever uma mensagem e exibi-la. Portanto, devemos alterar a *action* `index()`, que será o primeiro método a ser executado, configuração definida no arquivo `route.php`:

```
$routes->connect('/', ['controller' => 'Pets', 'action' => 'index']);
```

Vamos alterar a *action* em questão adicionando o seguinte código:

```
public function index()
{
    $this->response->withStringBody("Página em manutenção");
    return $this->response;
}
```

Em vez de um erro, exibiremos para o usuário uma mensagem mais adequada, embora simples, mas que exemplifica algumas das possibilidades que temos utilizando o objeto `response`.

6.7 Interagindo com views

Por padrão, o CakePHP, no final da execução de uma *action*, vai tentar renderizar uma *view*, que pode ser uma página para exibir adequadamente alguma informação. Você pode estar pensando: "Ah, agora vou aprender sobre *views*"! Bem, ainda não é o momento, porém é importante saber quais os mecanismos usados nos *controllers* para interagir com as *views*.

Todos os *templates* devem ser armazenados no diretório `/templates/`. Quando uma *action* é executada, por convenção, o template que será localizado tem o mesmo nome da *action* que recebeu a requisição. Por exemplo, se chamarmos a *action* `PetsController:index()`, a *view* que será processada deve estar localizada em `template\Pets\index.php`; caso contrário, se o arquivo `index.php` não existir, uma exceção do tipo `Cake\View\Exception\MissingTemplateException` será gerada.

> **Alguém lembra dos arquivos CTP?**
>
> Nas versões anteriores do CakePHP (sendo mais preciso, até a versão 3.8), os templates do CakePHP deveriam ser criados com a extensão **.ctp**, por exemplo, index.ctp, create.ctp, dashboard.ctp etc. CTP é uma abreviação da palavra *Cake template*. Quem trabalhou ou estudou o CakePHP antes da versão utilizada neste livro tem que utilizar essa extensão nos arquivos que representam *views*.

Quando uma página HTML é exibida para o usuário, o método que cria essa visualização é o `Cake\Controller\Controller::render(string $view, string $layout)`. Por padrão, esse método é chamado no final de cada *action* executada, a menos que este comportamento seja desabilitado no *controller*, o que pode ser feito atribuindo `false` à variável `$this->autoRender`, `$this->autoRender = false`. Podemos desabilitar a propriedade `autoRender` quando pretendemos renderizar

uma *view* sem nenhuma relação com o padrão estabelecido pelo CakePHP; logo, temos que informar para o método qual arquivo será utilizado, por exemplo:

```
$this->autoRender = false;

public function view()
{
    $this->render('/Custom/minha_pagina');
}
```

Observe que, como a opção `autoRender` está desabilitada, podemos definir qual arquivo será utilizado na *action* requisitada. No caso da *action* `view()` do exemplo citado, por convenção, o template `.php` que será utilizado deve estar localizado em `templates\Pets\view.php`; porém, com a mudança de configuração, o arquivo passado encontra-se em `template\Custom\minha_pagina.php`. Observe que o diretório base para os templates sempre é `templates`.

Criando páginas para o AdPET

Usando o nosso projeto, sabemos que atualmente temos duas rotas e duas *actions* que precisam de templates `.php` para visualizar as informações na tela do usuário. Se tentarmos acessar essas rotas atualmente, vamos lançar uma exceção `MissingTemplateException`, caso a resposta da *action* não seja tratada adequadamente com o `Response`.

Dando continuidade ao desenvolvimento, vamos criar esses arquivos e completar os arquivos necessários para a execução completa de um *request*. Incialmente, caso o servidor esteja parado, vamos logo rodar um `cake server` para subir a nossa aplicação.

Voltando ao *controller* `PetsController`, precisamos criar um diretório dentro de `templates`. Essa é mais uma convenção usada pelo CakePHP: os arquivos `.ctp` devem estar localizados dentro de um diretório que possui o mesmo nome do *controller* sem o sufixo `Controller`; logo, a classe `PetsController` requer um diretório

chamado `Pets` criando a estrutura `templates\Pets`. Dentro desse diretório, devemos criar um template para cada *action* que pode ser executada. Como definimos apenas duas rotas que criam um *request* para as *actions* `index()` e `view()`, vamos criar dois arquivos vazios, o `index.php` e o `view.php`. Se no browser entrarmos com a URL `http://localhost:8765/`, obteremos a seguinte visualização:

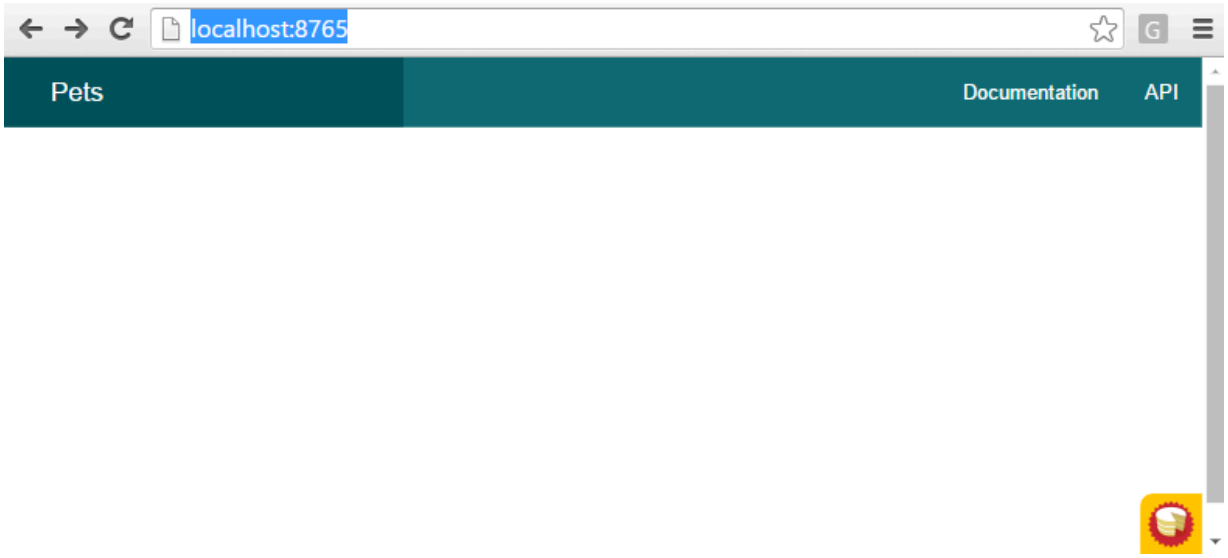


Figura 6.3: Visualização do arquivo index.ctp.

Para testar a segunda *action*, a URL é `http://localhost:8765/pet/1` — lembrando que o número **1** simboliza o `id` de um registro no banco de dados, porém aprenderemos adequadamente como acessar informações do banco de dados no próximo capítulo. Neste momento, se acessarmos essa URL, vamos receber uma exceção do tipo `Cake\Datasource\Exception\RecordNotFoundException`, que, neste caso, é porque ainda não temos as classes necessárias para acessar registros no banco de dados. Porém, é uma ótima oportunidade de apresentar outro recurso disponível nos *controllers* para desviar o fluxo para outra *action* ou site.

6.8 Redirecionando conteúdo

Todo *controller* dispõe do método `$this->redirect(string|array $url, integer $status)` para desviar o fluxo, redirecionar uma requisição para outra *action* ou redirecionar para uma URL de um site qualquer. Este método recebe dois parâmetros, uma URL e um código de status HTTP, por exemplo, para criar um redirecionamento para outra *action*, que pode ser do mesmo *controller* ou não. Podemos escrever o seguinte código:

```
return $this->redirect(['controller' => 'Report','action'=>'print']);

return $this->redirect(['action' => 'delete', $id]);

return $this->redirect('http://www.example.com');
```

Nesse código, temos um redirecionamento para o *controller* `ReportController` e a *action* `print`. No segundo exemplo, o fluxo será desviado para a *action* `delete()`, passando como parâmetro o `id`. Podemos, ainda, passar uma URL para o método, o que também vai funcionar.

Você pode estar se perguntando: "Como isso pode ajudar no problema anterior, que gera uma exceção do tipo `RecordNotFoundException`?". Simples, apenas vamos tratar essa exceção no código e redirecionar quando o erro for lançado. Primeiro, vamos importar a classe `RecordNotFoundException` com a declaração `use`:

```
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;


class PetsController extends AppController
{
    //continua
}
```

Agora podemos capturar a exceção no método `view` adicionando as cláusulas `try{}catch(){}`.

```
try{

    $pet = $this->Pets->get($id, ['contain' => []]);
}
catch(RecordNotFoundException $ex){

   return $this->redirect(['action' => 'index'], 404);
}
finally{

   $this->set('pet', $pet);
   $this->set('_serialize', ['pet']);
}
```

Pronto, agora esse trecho do código não vai mais parar quando algum problema relacionado ao registro do banco de dados lançar uma exceção do tipo `RecordNotFoundException`. Vamos capturar e redirecionar o fluxo de informação para a *action* `index()` e informar o código HTTP 404. No *controller*, existe também outro método de sintaxe mais simples para redirecionar para outras *actions* do mesmo *controller*, o método `$this->setAction($action, $args...)`. Mesmo sem todos os recursos do método `redirect()`, esse método também redireciona o fluxo para outra *action* com a possibilidade de passar ou não parâmetros. Para fixar mais o conteúdo, vamos reescrever a *action* `view()` usando o método `setAction()`. Temos:

```
try{

    $pet = $this->Pets->get($id, ['contain' => []]);
}
catch(RecordNotFoundException $ex){

   return $this->setAction('index');
}
finally{
```

```
    $this->set('pet', $pet);
    $this->set('_serialize', ['pet']);
}
```

Recapitulando, quando a intenção é criar algum redirecionamento, temos duas possibilidades: o `$this->setAction()`, bem mais recomendado quando a *action* alvo pertencer ao mesmo _controller; e o método `$this->redirect()`, mais complexo e que pode criar vários meios de redirecionamento.

6.9 Enviando dados para as páginas

Uma outra responsabilidade do *controller* é interagir com as *views*. Os dados que serão apresentados para a pessoa usuária são repassados pelos *controllers* para a camada de apresentação desenhada na arquitetura MVC.

A maneira que os *controllers* podem passar dados para a *view* é através do método `Controller::set()`, que basicamente cria variáveis que podem ser acessadas na *view* da *action* requisitada. Por exemplo, vamos usar novamente o método `index()` do nosso projeto de estudo para entender como funciona a passagem de valores.

No método, vamos passar a seguinte mensagem: "Aprendendo CakePHP". É uma mensagem bem simples, pois a intenção é aprender como enviar informação para a *view*. Neste caso, vamos escrever o seguinte código:

```
public function index()
{

  $this->set('message', 'Aprendendo CakePHP');
}
```

Nesse trecho, é criada uma variável chamada `$message`, que recebe como valor a mensagem "Aprendendo CakePHP". No template da

action — lembrando que o template, por convenção, tem o mesmo nome da *action*, mas são *lowercased* e *underscored*, ou seja, com o nome em minúsculo e com palavras separadas pelo símbolo sublinhado —, o nome do arquivo é `index.php`. Editando este arquivo, temos:

```
//src/Template/Pets/index.ctp

<h2><?php echo $message; ?></h2>
```

Então, no arquivo `index.php`, é possível imprimir o valor da variável `$message` definida na action `index()`. Se o browser for atualizado no endereço, vamos imprimir a mensagem como na imagem a seguir:

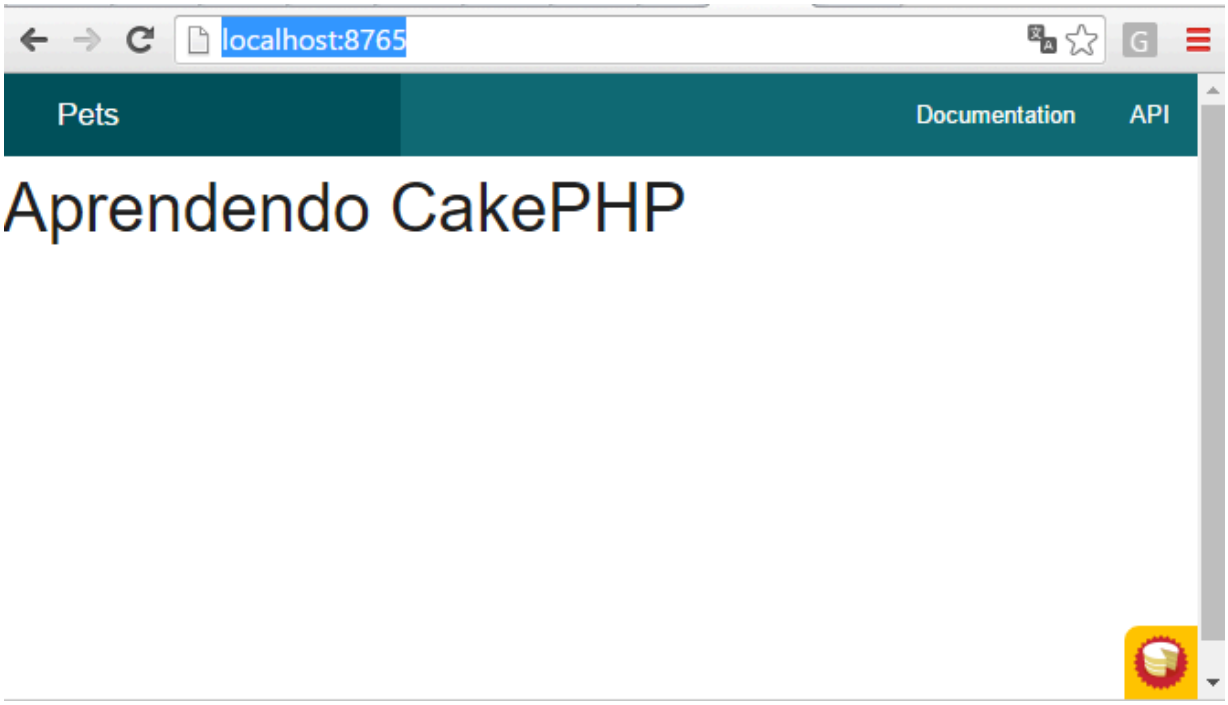


Figura 6.4: Visualizando uma mensagem.

Recapitulando, temos que criar templates `.php` com os mesmos nomes das *actions* requisitadas. Quando usamos o método `$this->set()` para criar valores, automaticamente a *view* vai poder acessar esses valores no arquivo `.php` que representa o template da *action* em ação. Guarde bem esses passos, pois eles serão a principal maneira para criar uma comunicação entre *controller* e *view*. Claro

que podemos usar, em alguns casos, alguma variável de sessão, porém abordaremos esse estudo mais adiante.

É possível passar também um *array* de valores para a *view*, facilitando a declaração de variáveis. Por exemplo, usando o mesmo código mostrado anteriormente, vamos adicionar mais uma variável, chamada `version`. Reescrevendo o método `index()`, o novo código será:

```
public function index()
{

  $dados = [
      'message' => 'Aprendendo CakePHP',
       'version' => '4.0'];

  $this->set($dados);
}
```

Com esse código, passando o *array* `$dados` para o método `$this->set($dados);`, estamos criando duas variáveis, `$message` e `$version`. No arquivo `index.php`, vamos fazer a seguinte modificação para exibir todos os valores definidos na *action* `index()`.

```
//src/Template/Pets/index.ctp

<h2><?php echo $message; ?> <?php echo $version; ?></h2>
```

Com esse código, temos a seguinte mensagem na página: **Aprendendo CakePHP 4.0**, uma junção das duas variáveis sendo exibidas na página. Através do controller, é bem simples passar informação para alguma *view*, podemos dizer também que é bem simples a maneira como o CakePHP trabalha de forma geral.

O controller Pet

No início do capítulo, criamos a classe `PetsController` usando o comando `cake bake controller`, que gerou uma classe com bastantes códigos, que nada mais são que códigos executando ações do

CRUD. No entanto, vamos agora deletar esses códigos e recriá-los mais adiante, pois ainda precisamos aprender sobre modelos para entender qual a lógica utilizada para criar as *actions* oriundas da execução do comando.

A estratégia de utilizar os métodos escritos pelo comando `bake` era para acelerar o aprendizado, apresentando algum código para ser manipulado e aprendendo novos conceitos. Então, vamos lá! Por enquanto, somente precisamos deixar o nosso *controller* como a estrutura exibida a seguir:

```
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;


class PetsController extends AppController
{
    public function index(){

    }

    public function view(){

    }
}
```

Pronto. Neste momento, é dessa estrutura que precisamos, pois ela é o esqueleto básico de qualquer *controller* que será criado mais adiante para atender a outras situações. Em `PetsController`, criamos duas *actions*, que são os métodos que serão chamados pelas rotas que criamos anteriormente. Uma *action* vai exibir uma lista de animais e a outra vai exibir os detalhes do animal quando for solicitado.

6.10 Eventos

Os *controllers* possuem eventos que são disparados em diferentes etapas do ciclo de vida da classe. Para cada evento, temos um respectivo *listener* que pode ser utilizado para executar alguma ação. Os eventos disponíveis são: `Controller.initialize`; `Controller.startup`; `Controller.beforeRedirect`; `Controller.beforeRender` e `Controller.shutdown`.

Para tratar os eventos, temos quatro *listeners*:

- `Cake\Controller\Controller::beforeFilter(EventInterface $event)`: é executado quando o evento `initialize` é disparado ou quando alguma *action* requisitada não existe no *controller*.
- `Cake\Controller\Controller::beforeRender(EventInterface $event)`: é chamado antes de uma *view* ser visualizada;
- `Cake\Controller\Controller::afterFilter(EventInterface $event)`: é o último método chamado no ciclo de vida de uma *action* e sempre é executado quando a renderização de uma *view* é finalizada;
- `Cake\Controller\Controller::shutdown(EventInterface $event)`: é o último método chamado no ciclo de vida de uma *action*, e sempre é executado quando a renderização de uma *view* é finalizada e o evento `Controller.shutdown` é disparado.

Vale lembrar que todo *controller* estende de `AppController` e é preciso chamar o *callback* também na superclasse para manter a relação de herança entre as classes. Exemplificando, temos:

```
namespace App\Controller;

use Cake\Event\EventInterface;

class PetsController extends AppController
{
    public function beforeFilter(EventInterface $event)
    {
```

```
        parent::beforeFilter($event);
    }

    public function beforeRender(EventInterface $event)
    {
        parent::beforeRender($event);
    }

    public function afterFilter(EventInterface $event)
    {
        parent::afterFilter($event);
    }
}
```

É bom ter em mente que em algum momento será necessário utilizar algum *callback* em alguma etapa do ciclo de vida do *controller*, seja para tratar alguma exceção ou apenas adicionar uma lógica complementar. Mais adiante, quando formos tratar sobre autenticação dos usuários, veremos um exemplo prático da utilização dos eventos no *controller*, no qual vamos usar o `beforeFilter` para determinar quais *actions* podem ser acessadas quando o usuário não está autenticado.

Conclusão

Neste capítulo, foram vistas muitas informações em relação ao *controller*. Vimos que os *controllers* são responsáveis por receber as requisições e enviar uma resposta depois que alguma lógica foi processada pelas *actions*. Também sabemos que o *controller* é o elo entre a camada de modelos e a camada de visualização e, portanto, possui métodos com funções bem específicas, como enviar informação para uma *view* ou carregar modelos que acessam informações no banco de dados.

Além disso, eles são capazes também de inicializar componentes, como paginação, controle de formulários, autenticação, entre outros. À medida que o conhecimento for avançando, vamos fazendo mais exemplos utilizando o projeto de estudo AdPET. No próximo capítulo,

estudaremos os *models* e aprenderemos como salvar, atualizar ou resgatar informações do banco de dados e exibi-las para o usuário.

Capítulo 7
Modelos

O modelo (ou *model*, em inglês), é a camada da arquitetura MVC usada para representar e manipular os dados, as regras de negócio da aplicação. É nela que existem meios para inserir, deletar, atualizar ou localizar informações. Um modelo não acessa a camada de visualização, ou seja, um modelo não deve apresentar dados, isso é função apenas da camada de visualização. O CakePHP tem em sua estrutura a arquitetura MVC bem representada. Apenas *controllers* têm mecanismos de acesso à camada de visualização e aos modelos, como apresentado no capítulo passado.

Em versões anteriores do framework, apenas uma classe era responsável por representar um modelo. A partir da versão 3.x do CakePHP, são necessárias duas classes: `Table` e `Entity`. Uma `Table` representa uma coleção de objetos enquanto uma `Entity` representa um único objeto. Ambas as classes trabalham em conjunto e fazem parte da nova arquitetura do ORM desenhada para o CakePHP para prover a manipulação dos dados.

O que é ORM

ORM, do inglês *Object Relational Mapping*, é uma técnica para criar uma camada que relacione o modelo de dados da aplicação com o banco de dados. Ou seja, as tabelas do banco de dados serão representadas por classes e os registros da tabela, por instâncias destas classes. Um ORM fornece mais velocidade no desenvolvimento, pois oferece mecanismos para que a pessoa desenvolvedora manipule o banco de dados da aplicação apenas trabalhando com classes e objetos em uma determinada linguagem de programação.

Cada tabela criada no banco de dados representa um modelo para o CakePHP. No capítulo 3, aprendemos como criar tabelas no banco de dados usando *migrations*; agora, vamos aprender a criar classes que representem essas tabelas para trabalhar com a ORM do CakePHP.

> **Convenção para classe dos modelos**
>
> Outro ponto importante que deve ser relembrado antes de começarmos a escrever código é a convenção adotada pelo CakePHP para nomear tabelas e colunas. Os nomes das tabelas e colunas devem ser escritos em inglês usando o padrão *CamelCase* e no plural. Por exemplo: `Pets`, `Users`, `Breeds` etc.

Caso você precise ou queira usar outro idioma para nomear os modelos, é necessário usar a classe `Cake\Utility\Inflector` para escrever novas regras e mudar a convenção padrão do framework para funcionar de acordo com o novo idioma. No entanto, neste livro, não será necessário mudar nada em relação ao idioma.

7.1 Classe Table

Uma classe `Table` deve ser criada para prover acesso a uma coleção de registros de uma determinada tabela no banco de dados. Toda classe `Table` deve ter uma respectiva tabela criada no banco de dados da aplicação cujo nome deve seguir a convenção adotada pelo CakePHP. Usando as convenções de nomenclatura adotadas, facilmente o Cake ORM consegue fazer o mapeamento entre essas duas estruturas.

Todas as classes `Table` devem ser criadas no diretório `src/Model/Table` e devem estender a classe `Cake\ORM\Table`. Quando o objetivo é listar, filtrar ou atualizar alguma informação no banco de dados da aplicação, é com esta classe que devemos trabalhar, pois

ela vai fornecer todos os métodos que permitem operações dessa natureza. A estrutura básica de uma classe `Table` é:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ModelNameTable extends Table
{
}
```

Lembre-se sempre de que o nome da classe também deve ser escrito em inglês e no plural, na mesma forma como são os nomes das tabelas.

Criando uma classe Table

Para entender melhor o funcionamento de uma classe `Table` e, claro, revisar um pouco de tudo o que foi visto nos capítulos anteriores, vamos usar nossa aplicação de estudo, pois agora podemos acrescentar um modelo e fechar um fluxo completo de uma requisição. No capítulo 3, foi abordada a criação de tabelas usando *migrations*. Uma das tabelas criadas foi a `Pets` , responsável por guardar as informações dos animais. Para relembrar, o comando para criar essa tabela foi:

```
$ bin/cake migrations create CreatePets
```

No método `change()` da *migration*, foi colocada toda a lógica para criar a tabela e as colunas, como descrito a seguir:

```
public function change()
{
    $table = $this->table('pets');

    $table->addColumn('name', 'string', [
        'default' => null,
        'limit' => 255,
        'null' => false,
```

```
        ]);
        $table->addColumn('description', 'text', [
            'default' => null,
            'null' => false,
        ]);
        $table->addColumn('gender', 'string', [
            'default' => 'M',
            'limit' => 2,
            'null' => false,
        ]);
        $table->addColumn('comment', 'text', [
            'default' => null,
            'null' => true,
        ]);
        $table->addColumn('birthday', 'datetime', [
            'default' => null,
            'null' => true,
        ]);
        $table->addColumn('created', 'datetime', [
            'default' => null,
            'null' => true,
        ]);
        $table->addColumn('modified', 'datetime', [
            'default' => null,
            'null' => true,
        ]);

        $table->create();
    }
```

Esse bloco de código é apenas para o método `change()`. Os códigos completos das migrations e das outras classes podem ser encontrados no GitHub do projeto, disponível em https://github.com/jozecarlos/adpet/.

Agora, caso não tenha feito anteriormente, precisamos executar essa *migration* para aplicar essas mudanças no banco de dados. Antes de executar, vamos verificar a configuração do banco de dados, que foi feita no capítulo 2 usando o `MySQL` — lembrando que

o arquivo que contém a configuração para definir qual é o banco de dados além das credenciais de acesso é o `app.php`, no diretório `config`. É necessário que a conexão com o MySQL esteja concluída com sucesso ou então não será possível usar *migration* para criar a tabela — e, claro, usar uma `Table` para listar os registros também não será possível. O comando para executar as *migrations* é:

```
$ bin/cake migrations migrate
```

Após a execução da *migration*, assumindo que o processo foi finalizado com sucesso, vamos iniciar a escrita da nossa classe `PetsTable.php`, referente à tabela `Pets` no diretório `src/Model`, com o código a seguir:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class PetsTable extends Table
{
}
```

Com a classe e a tabela criadas, já é possível trabalhar com dados da tabela `Pets`. Para testar nossa classe `PetsTable`, vamos precisar do *controller* `PetsController` e usar a *action* `index()`, que será utilizada para listar os registros da tabela `Pets` quando acessarmos o sistema. Vamos adicionar nesta *action* as seguintes linhas de código:

```
use Cake\ORM\TableRegistry;

public function index()
{
    $table_pets = TableRegistry::get('Pets');
    $query = $table_pets->find('all');
    $pets = $query->all();
    $this->set('_serialize', ['pets']);
}
```

Esse código utiliza uma instância da classe `PetsTable` para acessar os dados da *table* `Pets` no banco de dados. O objeto `$table_pets` é criado utilizando a classe `TableRegistry` em `TableRegistry::get('Pets')`. O método `find()` simplesmente retorna os registros de uma tabela. Porém, esse método não vai retornar de imediato os dados registrados na tabela, mas, sim, uma instância da classe `Cake\ORM\Query`, cujo objeto permite acrescentar mais condições para a pesquisa que será executada e filtrar os dados do resultado. Esse estudo será mais bem detalhado mais adiante.

> **QUERY BUILDER**
>
> A classe `Cake\ORM\Query` é uma nova abordagem inserida desde a versão 3.x do CakePHP, que permite a criação de *queries* mais complexas de uma forma mais programática e fluente. Com objetos desta classe, podemos decidir que condições entram ou não na consulta, dependendo da lógica em questão, antes que os dados sejam retornados.

Em seguida, para finalmente executar a consulta e obter os dados da tabela, é executado o método `all()`, que retorna uma coleção de objetos `Entity` (classe de estudo do próximo tópico) e, então, os dados são enviados para a *view* usando o método do *controller* `set()`. No bloco de código da *action* `index()`, no exemplo descrito, estamos apenas listando tudo o que está na tabela sem uso algum de cláusulas condicionais, mas, no decorrer da leitura, aprenderemos como criar consultas mais elaboradas.

Para finalizar exibindo as informações para o usuário, por convenção, a *view* que será exibida usa o arquivo `index.php`. Para facilitar, usaremos uma tabela HTML para representar as informações na página. Podemos escrever o código HTML do template com o seguinte código:

```
<h2>Pets List</h2>
<table>
```

```
    <thead>
        <tr>
            <th>Name</th>
            <th>Birth Day</th>
            <th>Description</th>
        </tr>
    </thead>
    <tbody>
        <?php foreach ($pets as $pet): ?>
          <tr>
              <td><?php echo $pet->name; ?></td>
              <td><?php echo $pet->birthday; ?></td>
              <td><?php echo $pet->description; ?></td>
          </tr>
        <?php endforeach; ?>
     </tbody>
</table>
```

A variável `$pet` dentro do *loop* `foreach`, relembrando, é declarada na action `index()` em `$this->set('_serialize', ['pets']);`. Esse é um pequeno exemplo que interliga todas as camadas MVC do CakePHP que recebem uma requisição; assim, podemos acessar dados da base e apresentar informações para o usuário. A seguir, já que agora sabemos o que é usado para manipular uma coleção de registros, aprenderemos como manipular registros individualmente.

7.2 Classe Entity

Falamos que, para representar uma tabela ou uma coleção de registros, usamos a classe `App\Model\Table`. Neste momento, imagino que você já pensou na seguinte questão: mas qual classe representa apenas uma tupla da tabela, seja para atualizar alguma informação ou apenas acessar informação? Bem, a resposta é que vamos precisar de um objeto da classe `Cake\ORM\Entity`, que tem a função de representar um registro da tabela. Todas as classes `Entity`

devem ser criadas no diretório `src/Model/Entity` como definido na convenção do CakePHP. A estrutura básica da classe é:

```
// src/Model/Entity/Exemplo.php
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Exemplo extends Entity
{
}
```

Na seção anterior, quando fizemos um exemplo para aprender como trabalhar como uma `Table`, implicitamente também estávamos trabalhando com uma `Entity`. Você deve estar se perguntando: "Como assim?". A resposta é simples, vamos rever o código:

```
$table_pets = TableRegistry::get('Pets');
$query = $table_pets->find('all');
$pets = $query->all();
$this->set('_serialize', ['pets']);
```

Quando executamos o método `all()` do objeto `$query`, criado com o uso do `find('all')`, é retornada uma lista de instâncias da classe `Entity`. O ORM do CakePHP, por padrão, cria um objeto para cada linha da tabela que é retornada na execução da consulta.

Por isso, no arquivo `index.ctp`, quando percorremos a variável `$pets`, podemos acessar as propriedades do objeto que correspondem a cada coluna criada na tabela `Pets`; logo, quando escrevemos `$pet->name;`, estamos lendo o valor retornado da coluna `name` de uma tupla da tabela. Se for preciso, por exemplo, acessar o valor da coluna `id`, é só escrever `$pet->id;`.

A criação de uma `Entity` não é obrigatória quando utilizamos as propriedades e funções padrão da classe, o exemplo anterior funcionaria normalmente sem ter criado o arquivo para a classe `Pet`. Quando é preciso reescrever algum comportamento da classe `Entity` ou criar alguma lógica mais personalizada para o modelo, temos que

criar a classe em `App\Model\Entity`. No caso da tabela `Pets`, a classe correspondente é:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Pet extends Entity
{
}
```

Uma `Entity` deve estender a superclasse `Cake\ORM\Entity` e a nomenclatura da classe está no singular justamente para reforçar a ideia de que representa um único registro.

7.3 Manipulando Entities

Existem duas maneiras de obter objetos de uma classe `Entity`. Podemos instanciar uma classe exatamente como aprendemos nos conceitos de Orientação a Objetos. Por exemplo, usando a classe `Pet`, temos:

```
use App\Model\Entity\Pet;

$pet = new Pet();
```

Vale lembrar que, para este caso, é obrigatório criar a classe `Pet` no diretório `App\Model\Entity`. O construtor da classe é `__construct( array $properties [] , array $options [] )` e possui dois parâmetros não obrigatórios. O primeiro é um *hashed array*, utilizado caso seja preciso criar um objeto com algumas propriedades com valores atribuídos, como:

```
$pet = new Pet([
    'name' => 'Supino',
```

```
    'birthday' => new DateTime('2016-01-01')
]);
```

O segundo parâmetro é um *array* com opções que determinam algum comportamento quando a `Entity` é criada. As opções disponíveis são:

- `setter` : determina se a atribuição de valores das propriedades usa ou não os métodos *setters* da entidade;
- `markClean` : embora os valores das propriedades sejam inicializados, caso seja usada esta opção, a entidade vai informar mesmo assim que as propriedades estão limpas.
- `markNew` : informa que a instância desta entidade não foi persistida ainda.
- `guard` : se usada, previne que propriedades inacessíveis sejam alteradas. O valor *default* é `false` .
- `source` : informa qual repositório a entidade representa.

A outra maneira de criar uma `Entity` é usando a classe `TableRegistry` , já usada anteriormente para criar um objeto da classe `PetsTable` e que também pode criar *entities* usando o método `get()` e `newEmptyEntity()` em conjunto, como demonstrado a seguir:

```
use Cake\ORM\TableRegistry;

$pets = TableRegistry::get('Pets')->newEmptyEntity();
```

O método `get( string $alias , array $options [] )` recebe como parâmetro o nome da tabela e retorna um objeto de `Cake\ORM\Table` que possui o método `newEmptyEntity()` . É possível reescrever o mesmo código da seguinte maneira:

```
use Cake\ORM\TableRegistry;

$pets_table = TableRegistry::get('Pets');
$pets = $pets_table->newEmptyEntity();
```

Nesse código, a única diferença do anterior é que estamos guardando o retorno do método `TableRegistry::get('Pets')` antes de

chamar `newEmptyEntity()`. É uma mudança que permite reutilizar o objeto `$pets_table` para outras finalidades.

Um objeto `Entity` pode ser criado também com algumas propriedades inicializadas com o método `newEntity()`, se for passado um *array* de dados onde a chave referencia uma propriedade, como no exemplo a seguir:

```
use Cake\ORM\TableRegistry;

$pets_table = TableRegistry::get('Pets')->newEntity([
    'name' => 'Supino',
    'birthday' => new DateTime('2016-01-01')
]);
```

Esses são exemplos de código de como criar objetos, que serão úteis quando começarmos a salvar informações no banco de dados. No momento, peço para que o leitor ou leitora aprenda essas duas maneiras, pois vamos utilizá-las mais adiante quando estivermos trabalhando com formulários.

7.4 Métodos get() e set()

Objetos do tipo `Entity` disponibilizam os métodos `get()` e `set()` para cada propriedade que ele possui. Para ler informações, podemos usar o método `get()` no exemplo citado anteriormente para imprimir informações da classe `Pet` no arquivo `index.ctp`. Poderíamos reescrever usando o `get()` e o resultado seria:

```
    <tbody>
        <?php foreach ($pets as $pet): ?>
          <tr>
              <td><?php echo $pet->get('name'); ?></td>
              <td><?php echo $pet->get('birthday'); ?></td>
              <td><?php echo $pet->get('description'); ?></td>
          </tr>
```

```
        <?php endforeach; ?>
    </tbody>
```

O método `set()` é utilizado quando a ação é passar um valor para a propriedade. Por exemplo: eu quero criar um objeto `Pet` e atribuir um valor para a propriedade `name`; nesse caso, temos:

```
use App\Model\Entity\Pet;

$pet = new Pet();
$pet->set('name', 'Supino' );
```

Sempre que usamos o método `set()`, temos que informar o nome da propriedade no primeiro parâmetro do método e o valor, no segundo. O método ainda tem uma peculiaridade: é possível atribuir valores para um grupo de atributos em uma única execução. A seguir, veja um exemplo de como podemos escrever o código atribuindo valor para propriedade `name` e `birthday`:

```
use App\Model\Entity\Pet;

$pet = new Pet();

//Passando valor para o atributo name
$pet->set('name', 'Supino' );

//Passando valor para o atributo name e birthday
$pet->set([ 'name' => 'Supino', 'birthday' => new DateTime('2016-01-01')
]);
```

Logo, quando queremos atribuir valores para mais de um atributo, devemos passar um *hashed array* como parâmetro contendo todos os elementos e valores que desejamos. É um artifício útil e que evita criar uma linha de código para cada atributo da classe, simplificando o código escrito.

Embora uma `Entity` disponibilize métodos públicos como `get()` e `set()` para acessar ou atribuir valores, ainda assim é possível fazer as mesmas ações acessando diretamente uma propriedade. Isso é

possível porque a classe `Entity` implementa os métodos de sobrecarga definidos pelo POO do PHP para acessar ou atribuir valores nas propriedades, que são: `__set()`, `__get()`, `__isset()` e `__unset()`.

O uso dos métodos `get()` ou `set()` não é obrigatório e fica a critério do desenvolvedor ou da desenvolvedora decidir qual é a melhor maneira de acessar as propriedades, pois ambas são seguras e corretas. Resumindo, para cada propriedade, podemos acessar como é escrito no exemplo a seguir:

```
use App\Model\Entity\Pet;

$pet = new Pet();

$pet->set('name', 'Supino' );
$pet->name = 'Supino';

echo $pet->get('name');
echo $pet->name;
```

7.5 Accessors e Mutators

Imagine que durante o desenvolvimento surja a necessidade de mudar o valor da propriedade de um objeto `Entity`, seja na leitura ou na atribuição de valor. Uma `Entity` permite a alteração do comportamento de um atributo quando é lido ou atualizado por meio de métodos chamados de *accessors* ou *mutators*.

Para criar um *accessor*, use o prefixo `_get` junto do nome do atributo, que é escrito usando o formato CamelCase. Por exemplo, na classe `Pet`, queremos mudar o comportamento da propriedade

`name`, então a assinatura do *accessor* será `_getName($name)`, como apresentado no código a seguir:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Pet extends Entity
{
    protected function _getName($name)
    {
        return strtoupper($name);
    }
}
```

Esse código formata para maiúsculo o valor do atributo `name` quando ele for lido em um determinado momento. Ou seja, o valor natural da propriedade `name` é modificado para caracteres maiúsculos durante a leitura no exemplo anterior. Um método *accessor* é utilizado quando pretendemos customizar um valor, porém esse valor que foi alterado será salvo no banco de dados quando alguma operação para persistir a `Entity` for executada.

Logo, para esse caso apresentado, quando um novo registro for salvo na tabela `pets`, o *accessor* definido na classe `Pet` vai modificar o valor da propriedade `name`, o qual será persistido com os caracteres em maiúsculo no banco de dados.

Quando a intenção é alterar um atributo sem que essa mudança defina como o valor será definitivamente persistido no banco de dados, usamos um *mutator*. Basicamente, um método *mutator* vai alterar como um valor é lido pelos métodos `get()` e `set()` e também no caso de acesso direto ao atributo do objeto.

A convenção criada pelo CakePHP para criar um *mutator* é usar o prefixo `_set_` junto do nome do atributo, que deve ser escrito no formato CamelCase e sempre deve retornar um valor. Por exemplo, um *mutator* para o atributo `name` poderia ser:

```php
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Pet extends Entity
{
    protected function _setName($name)
    {
        return "Pet: ".$name;
    }
}
```

7.6 Campos virtuais

As propriedades de uma `Entity` são criadas tendo como referências as informações que são salvas no banco de dados, isto é, os campos de uma tabela serão representados pelas propriedades de uma `Entity`. Um campo virtual é uma propriedade que não representa um campo de uma tabela, mas tem seu valor definido dinamicamente por alguma lógica escrita na `Entity`.

Para criar um campo virtual, devemos usar o prefixo `_get` junto do nome da propriedade escrita usando o formato CamelCase. Por exemplo, na classe `Pet`, temos a propriedade `birthday`, que armazena uma data representando o dia de nascimento do animal. Usando essa informação, poderíamos criar um campo virtual para informar a idade do animal chamado `age` cujo valor não existe em nenhum campo da tabela `pets`. Escrevendo este campo virtual, o código seria:

```php
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Pet extends Entity
{
```

```
    protected function _getAge()
    {
        return $this->calcularIdade($this->birthday);
    }

    private function calcularIdade( $dataDeNascimento ){
       if (!empty($dataDeNascimento)) {
            return (intval(date('Y', time() -
strtotime($dataDeNascimento))) - 1970) + 1;
        }
        return 'Sem idade definida - 0 ';
    }
}
```

O método `_getAge()` cria um campo/propriedade virtual que pode ser acessada por uma instância da classe `Pet` — por exemplo, `$pet->age` . A função `calcularIdade( $dataDeNascimento )` é um método auxiliar para calcular a idade usando uma data como parâmetro, o corpo deste método não é importante para ser exibido neste tópico porque foi utilizado apenas para fins didáticos. Complementando o exemplo que iniciamos, vamos usar o arquivo `index.ctp` , já utilizado anteriormente, e usar este campo virtual para exibir a idade do animal. Então, o código com essa modificação será:

```
<h2>Pets List</h2>
<table>
    <thead>
        <tr>
            <th>Name</th>
            <th>Birth Day</th>
            <th>Description</th>
            <th>Age</th>
        </tr>
    </thead>
    <tbody>
        <?php foreach ($pets as $pet): ?>
          <tr>
              <td><?php echo $pet->name; ?></td>
              <td><?php echo $pet->birthday; ?></td>
```

```
            <td><?php echo $pet->description; ?></td>
            <td><?php echo $pet->age; ?></td>
        </tr>
    <?php endforeach; ?>
</tbody>
</table>
```

Quando um campo virtual é definido, qualquer objeto de uma `Entity` vai ter acesso a essa propriedade — fato observado no exemplo anterior, em que a variável `$pet` é um objeto da classe `Pet` e, por isso, pode acessar a propriedade `age`. Uma observação importante: um campo virtual não pode ser usado nos métodos de pesquisa como o `find()` justamente porque essas propriedades não representam um campo na tabela.

7.7 Mass assignment

Usamos o `mass assignment` para atribuir valores para um grupo de propriedades na inicialização de `Entity`, o que pode ajudar a melhorar a legibilidade do código e acelerar o desenvolvimento. Qual programador não deseja que isso aconteça? Porém, usar o *mass assignment* pode também inserir alguns problemas, como fazer com que algumas propriedades que não devem ser inicializadas em massa sejam inicializadas. Para entendermos melhor, uma atribuição em massa basicamente seria:

```
use App\Model\Entity\Pet;

$pet = new Pet(['id' => 1, 'name' => 'Rock']);
```

Nesse exemplo, estamos atribuindo valores para `id` e `name` na construção do objeto `$pet`. É notável que a construção de um objeto fica mais fácil e rápida, porém o uso do `mass assignment` pode inserir vulnerabilidades na aplicação pelo fato de conseguir atribuir valores para todas as propriedades, caso não sejam especificadas

quais propriedades devem ser inicializadas com este comportamento.

Digamos que propriedades de uso interno da aplicação sejam modificadas por um simples `Request` do usuário. Caso isso aconteça, a aplicação pode ter um comportamento inesperado e errado. Para configurar quais propriedades podem ser inicializadas via `mass assignment`, devemos especificar na *Entity* usando a propriedade `$_accessible`, por exemplo:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Pet extends Entity
{
    protected $_accessible = [
        'name' => true,
        'birthday' => true,
        '*' => false
    ];
}
```

Nesse código, somente as propriedades `name` e `birthday` serão alteradas caso o `mass assignment` seja usado na classe `Pet`. O campo `*` determina que o restante das outras propriedades não pode ser inicializado em massa. Esse comportamento ainda pode ser alterado com o método `setAccess()`. Digamos que, na sua lógica, tenha uma condição que necessite mudar o comportamento definido inicialmente, por exemplo, `$pet->setAccess('user_id', true);`, assumindo que `user_id` é uma propriedade em que o `mass assignment` foi definido como `false`. Veja alguns exemplos a seguir:

```
    $pet->setAccess('user_id', true);
    $pet->setAccess('name', false);
```

Conclusão

Neste capítulo, aprendemos como utilizar as classes `Entity` e `Table`, estruturas base da ORM do CakePHP, por meio de conceitos e fundamentos que permitem a você saber exatamente qual a função delas no sistema. No próximo capítulo, será apresentado como consultar o banco de dados usando CakePHP ORM.

Capítulo 8
Queries

A camada de modelo do CakePHP tem mecanismos para representar eficientemente as informações do banco de dados usando classes que estudamos anteriormente, `Table` e `Entity`. E é também com essas classes que podemos carregar as informações do banco de dados a partir de métodos que criam consultas, sem precisar escrever nenhuma linha de código SQL. Ou seja, são classes que também fornecem toda a estrutura necessária quando a intenção é criar consultas para o banco de dados.

8.1 Criando consultas

Uma maneira rápida de consultar alguma informação no banco de dados é usando o método `Cake\ORM\Table::find($type, $options = [])`. Imagine que queremos listar os registros da tabela `pets`. Com SQL, seria uma consulta simples, como `SELECT * FROM pets`. Já com o uso dos modelos do CakePHP, essa mesma consulta seria:

```
$query = TableRegistry::getTableLocator()->get('Pets')->find();
$result = $query->all();
```

Como você pode ver, criar uma consulta com CakePHP é ainda mais simples, e usamos apenas PHP. Isso facilita para desenvolvedores, que não precisam escrever consultas em outra linguagem, fugindo do contexto da programação do projeto.

Na primeira linha do exemplo, não estamos recebendo os registros do banco de dados na variável `$query`, mas iniciando uma consulta. O objeto `$query` é uma instância da classe `Cake\ORM\Query` — este é um artifício importante e necessário para permitir a desenvolvedores

um passo a mais para filtrar os resultados antes de receber os registros do banco de dados.

Nesse exemplo, os registros de fato são armazenados na variável `$result`, que é uma instância da classe `Cake\ORM\ResultSet`, depois da execução do método `all()`. Para entender a importância do `ORM\Query`, no mesmo exemplo, imagine que queremos filtrar os resultados da tabela `pets` que pertençam ao usuário com `id` igual a `1`. Se você estiver habituado com SQL, saberá que é preciso usar a cláusula `where`, então a nova consulta seria `SELECT * FROM pets WHERE user_id =1`.

No CakePHP, a responsabilidade de adicionar cláusulas condicionais na consulta é da classe `Cake\ORM\Query`. Por isso, sempre que iniciamos uma consulta, primeiro temos que trabalhar com um objeto `Query` e adicionar, se preciso, alguma condição para filtrar os resultados e então receber os dados. Para esta mesma consulta, usando `where` com CakePHP seria:

```
$query = TableRegistry::getTableLocator()->get('Pets')->find();
$query->where(['user_id =' => 1]);

$result = $query->all();
```

Esse código, quando executado contra a base de dados, terá a mesma resposta da consulta feita com o SQL que abordamos como exemplo. Ou seja, para a cláusula `where`, a classe `Cake\ORM\Query` oferece um método `where()`, lembrando que `user_id` é uma coluna no banco de dados que associa o registro da tabela `pets` com a tabela `users`. Abordaremos associação entre tabelas e como trabalhar com elas no CakePHP mais adiante.

Uma vez que temos criado na nossa lógica um objeto de `Cake\ORM\Query`, podemos usá-lo para adicionar mais condições e montar *queries* mais complexas, por exemplo:

```
$query = $this->Pets->find()
    ->where(['Pets.created >' => new DateTime('-10 days')])
```

```
    ->limit(10);

$result = $query->all();
```

Uma peculiaridade observada nesse trecho de código é que o método `limit()` é chamado a partir do `where`. Isso somente é possível porque a classe `Query` implementa o padrão de projeto `Fluent Interface`, que permite encadear métodos a partir de outros métodos. Esse comportamento melhora a legibilidade do código e menos linhas são necessárias para escrever uma instrução.

A referência `$this->Pets` é uma instância da classe `Table`, que é criada automaticamente pelo *controller*. Pelo exemplo anterior, podemos afirmar que esse trecho de código só funcionaria em uma `action` de um *controller*. O método `find()` sempre vai retornar uma objeto `Cake\ORM\Query`, o qual vai permitir encadear um conjunto de outros métodos a fim de formar uma consulta mais elaborada. Dentre os métodos que podemos chamar, podemos destacar:

- `contain()`: carrega informações das tabelas associadas;
- `limit()`: define a quantidade de registros retornados na consulta;
- `order()`: ordena os registros;
- `select()`: determina quais propriedades do modelo serão incutidas no resultado;
- `distinct()`: remove registros repetidos do retorno da consulta;
- `group()`: útil para funções de agregação, adiciona uma cláusula `GROUP BY` na consulta;
- `having()`: adiciona uma cláusula `HAVING` à sua consulta.;
- `join()`: retorna a quantidade de registros consultados.

Todos esses métodos podem ser concatenados para formar uma consulta maior, com mais condições de pesquisa. Por exemplo, imagine que pretendemos selecionar os resultados na tabela `pets` agrupando-os pela data de criação, que o valor da coluna `views` seja maior que 3 e que apenas os valores das colunas `name`, `views` e

`description` retornem nos resultados. Transformando essa narrativa em código, temos:

```
$query = $this->Pets->find()
->select(['name', 'views', 'description'])
->group('created_at')
->having(['views >' => 3]);
```

É notável que os métodos espelham as cláusulas do `SQL` nativo, pois é uma característica que foi implementada no CakePHP intencionalmente. A pessoa desenvolvedora pode trazer todo o conhecimento adquirido com consultas no banco de dados para o projeto, mas escrevendo com sintaxe de código em PHP. Portanto, é claro que, embora o framework facilite bastante a elaboração de consultas, é importante que se tenha bons conhecimentos nesta área, assim, mais fácil e mais fluida será a utilização do `ORM` do CakePHP.

E o método `all()` ? Já sabemos que os dados só serão retornados quando esse método é executado, ou seja, quando a consulta é de fato disparada contra o banco de dados. Podemos dizer que um objeto `Query` é um objeto *lazy*, preguiçoso, e depende da execução de determinados métodos para retornar a consulta esperada. Um desses métodos é o `all()` , que aprendemos nos exemplos deste tópico, mas existem outras opções que podemos usar para executar uma consulta e que também mudam a forma como os dados são retornados. As opções são:

- `foreach()` : *loop* para percorrer os objetos da *query*;
- `first()` : retorna o primeiro resultado da consulta;
- `execute()` : usado com operações de `insert` , `update` e `delete` ;
- `toArray()` : retorna os resultados da consulta como um `array` ;
- `toList()` : comportamento semelhante a `toArray()` ;
- `all()` : retorna os resultados quando a operação são apenas consultas;
- `count()` : retorna a quantidade de registros consultados.

Entre esses métodos, o `foreach` é o único que não faz parte do conjunto de implementações do CakePHP, mas é uma instrução nativa do PHP que permite percorrer os itens de uma coleção. Quando aplicado sobre um objeto `Query`, também resolve a consulta percorrendo cada objeto do retorno. Por exemplo:

```
$query = TableRegistry::getTableLocator()->get('Pets')->find();

foreach ($query as $row) {
    debug($row->name);
}
```

Com o `foreach`, não é preciso executar o `all()`; ele mesmo é uma maneira de executar a consulta ao mesmo tempo que interage sobre os objetos dela. É um recurso útil se for preciso acessar cada item retornado e encaixar em alguma lógica que o desenvolvedor venha a criar.

Outros dois métodos que também quero comentar é o `toList()` e o `toArray()`. Eles têm um comportamento semelhante: transformam o retorno de uma consulta, que é um objeto `Cake\ORM\ResultSet`, em um simples `array`, onde cada elemento do *array* é uma instância da classe `Entity` que representa aquele registro no banco. Entenda melhor o `toArray()` no exemplo a seguir:

```
$resultArray = $this->Pets->find()->toArray();

$resultArray = [
    1 => object(App\Model\Entity\Pet) ,
    2 => object(App\Model\Entity\Pet) ,
];
```

O comportamento desse método ainda pode ser modificado se o método `find` receber como parâmetro o valor `list`; então, o retorno da consulta é transformado em um `array` de dados cuja chave é o valor da chave primária da tabela de cada registro, e o valor é a informação da propriedade que foi definida pelo método `displayField` na classe `Table` correspondente ao modelo. Nesse

exemplo, a classe na qual definiremos essa configuração é a `PetsTable`. Vejamos o código a seguir.

```
<?php

namespace App\Model\Table;

use Cake\ORM\Table;

class PetsTable extends Table
{
    public function initialize(array $config)
    {
        parent::initialize($config);
        $this->setDisplayField('name');
    }
}
```

Com essa configuração no modelo, ao executarmos o código a seguir, teremos como resposta um `array` bem mais simples, desidratado:

```
$resultArray = $this->Pets->find('list')->toArray();

$resultArray = [
    1 => 'Supino',
    2 => 'Layla',
];
```

Os valores `Supino` e `Layla` são os valores vindos da coluna `name` na tabela `pets`, e `1` e `2` são os valores da chave primária da tabela. O comportamento do `toList()` é semelhante a tudo que comentamos até o momento. A única diferença é que cada chave do `array` não representa o valor da chave primária da tabela, mas apenas uma lista numérica. O mesmo código reescrito com `toList()` teria como retorno:

```
$resultArray = $this->Pets->find('list')->toList;
```

```
$resultArray = [
    0 => 'Supino',
    1 => 'Layla',
];

$resultArray = $this->Pets->find()->toList;

$resultArray = [
    0 => object(App\Model\Entity\Pet) ,
    1 => object(App\Model\Entity\Pet) ,
];
```

Os valores são os mesmos, mas a chave do `array` representa uma posição numérica da lista.

Obtendo um único registro

A aplicabilidade do método `find()` é a de criar uma consulta com argumentos para, assim, trazer do banco de dados uma coleção de registros ou localizar um seleto grupo de registros baseado nas condições declaradas. No entanto, se tivermos em posse do `id` do registro no banco de dados, temos um método mais prático para retornar as informações do banco, que é usando o `Cake\ORM\Table::get($id, $options = [])`. Vejamos um exemplo:

```
$petTable = TableRegistry::getTableLocator()->get('Pets');
$pet = $petTable->get(1);
```

A variável `$pet` é uma `Entity` com as informações formadas a partir do registro na tabela `pets` com `id` igual a `1`. É uma chamada mais direta e que retorna imediatamente as informações do banco de dados. Um detalhe que a pessoa desenvolvedora precisa ter em mente é que a utilização desse método está sujeita a lançar uma exceção do tipo `Cake\Datasource\Exception\RecordNotFoundException` caso o `id` passado para o método não exista no banco de dados. Nesse caso, é uma boa prática envolver a lógica com o bloco de código `try/catch`, tratar de forma adequada esse problema e evitar que

uma mensagem de erro desagradável seja exibida para o usuário. Para ficar mais claro, deixo como sugestão o código a seguir:

```
use Cake\Datasource\Exception\RecordNotFoundException;

    /**
     * View method
     *
     * @param string|null $id Pet id.
     * @return void
     */
    public function view($id = null)
    {
        try{

            $petTable = TableRegistry::getTableLocator()->get('Pets');
            $pet = $petTable->get(1);
             $this->set('pet', $pet);
        }
        catch(RecordNotFoundException $ex){

           $this->redirect(['action' => 'index'], 404);
        }
    }
```

Nesse exemplo, caso o erro do tipo `RecordNotFoundException` seja capturado, redirecionamos para a `action index` do mesmo *controller*.

8.2 Ordenando resultados

Uma consulta geralmente precisa ordenar os resultados. É uma forma de organizar as informações para facilitar a exibição e o entendimento. Até este momento, já sabemos que a manipulação dos resultados de uma consulta pode ser feita com um objeto da classe `Cake\ORM\Query` . A ordenação dos dados pode ser tratada com o método `order()` , que recebe como parâmetro um *array* com as

propriedades que podem ser ordenadas de forma crescente (para as quais usamos o `ASC`) ou decrescente (com o `DESC`).

Podemos usar o exemplo anterior, no qual listamos todos os registros da tabela `pets` para ordenar os resultados pelo `name` de forma decrescente. O código reescrito teria o seguinte formato:

```
$query = TableRegistry::getTableLocator()->get('Pets')->find();
$query->where(['user_id =' => 1)->order(['name' => 'DESC']);

$result = $query->all();
```

Obtendo o primeiro registro de uma consulta

Quando precisamos obter o primeiro registro de uma consulta, temos que usar o método `first()`. Esse método sempre vai retornar o primeiro registro, independente das condições declaradas na consulta. Vamos exemplificar: digamos que você queira selecionar o primeiro registro da tabela `pets` ordenado pela coluna `name` de forma ascendente. Para resolver essa questão, o código seria:

```
$query = $this->Pets->find();
$query->order(['name' => 'ASC']);

$result = $query->first();
```

A pessoa desenvolvedora também precisa estar atenta, pois o valor retornado pelo `first()` pode ser um valor nulo, e isso pode gerar um erro na execução do código se não for tratado adequadamente.

8.3 Contando os resultados

Durante o desenvolvimento de um sistema, é bem comum precisar ter o conhecimento da quantidade de registros em uma tabela. Isso pode ser útil para encaixar em alguma lógica, como paginação, ou para alertar o usuário de alguma restrição. Com o CakePHP, para

saber a quantidade de registros de um determinado conjunto de dados, temos o método `count()`, que retornará um valor numérico inteiro. Uma observação importante é que, quando utilizamos o `count()`, não precisamos utilizar cláusulas como `limit`, `offset` ou `page`, pois elas são automaticamente ignoradas pelo método para obter o resultado final. Caso fossem executadas pelo `count()`, isso retornaria um valor divergente da contagem real dos registros da consulta.

Veja um exemplo de código de como utilizar o `count()`:

```
$query = TableRegistry::getTableLocator()->get('Pets')->find();
$number = $query->where(['breed_id =' => 2)->count();
```

Nesse exemplo, a consulta retorna a contagem dos registros para a variável `$number`, cujo valor é a soma dos registros da tabela `Pets` onde `breed_id` é igual a 2. O objeto `$query` usado para executar o `count()` ainda pode ser utilizado posteriormente para receber os registros que representam aquela quantidade, como no exemplo a seguir:

```
$query = TableRegistry::getTableLocator()->get('Pets')->find();
$number = $query->where(['breed_id =' => 2)->count();
$result = $query->where(['breed_id =' => 2)->all();
```

Agora, vamos pensar: se na chamada do `count()` do exemplo anterior fosse adicionada a instrução `limit(10)`, qual seria o possível resultado no retorno do método? Bem, já sabemos que em nada iria influenciar o retorno, então podemos afirmar que os valores das variáveis `$numberX` e `$numberY` no exemplo a seguir são, rigorosamente, os mesmos:

```
$query = TableRegistry::getTableLocator()->get('Pets')->find();
$numberX = $query->where(['breed_id =' => 2)->limit(10)->count();
$numberY = $query->where(['breed_id =' => 2)->count();
```

Podemos observar que o retorno da execução do `count()` ignora cláusulas como `limit`, mas não ignora cláusulas como o `where`, por exemplo. Ou seja, nos exemplos que utilizamos, o valor da

quantidade retornado está relacionado com a condição imposta, `breed_id =' => 2`, que representa a quantidade de registros na tabela que têm essa informação associada.

Outra questão que apresento é: sabendo que o resultado de uma `query` é uma coleção de resultados, poderíamos saber a quantidade de registros dessa coleção percorrendo cada item e aplicando uma lógica para armazenar a contagem? Bem, a resposta é "sim", mas eu aconselho você a não fazer isso. Um banco de dados vai sempre contar registros mais rápido do que executar esse tipo de operação em um *controller*. Como exemplo, deixo essa dica: tente ao máximo delegar a seleção de informações e a contagem de registros da sua lógica de negócio para o banco de dados.

8.4 Funções SQL

O SQL fornece funções para facilitar a criação de consultas mais complexas. Um exemplo de função no SQL é a `COUNT` — que, como já vimos, é representada no ORM do CakePHP pelo método `count()`. Além dessa função, o SQL fornece outros tipos de funções, como para calcular média, para retornar o valor máximo de um grupo de valores, entre outras. No CakePHP, temos suporte a todas as funções do SQL através do método `func()`. Por exemplo, para calcular a média dos valores da coluna `views` da tabela `pets`, temos que usar a função `AVG`. Com o CakePHP ORM escreveremos:

```
$query = $this->Pets->find();
$query->select(['count_avg' => $query->func()->avg('views')]);
```

Observando o código, vemos que a chave para acessar as funções é o método `func()`, que é uma instância da classe `FunctionsBuilder`, disponível através de um objeto da classe `Cake\ORM\Query`. As funções suportadas pelo ORM são:

- `avg($expression, $types = [])`

- `rand()`
- `sum($expression, $types = [])`
- `min($expression, $types = [])`
- `max($expression, $types = [])`
- `count($expression, $types = [])`
- `coalesce($args, $types = [])`
- `concat($args, $types = [])`
- `dateDiff($args, $types = [])`
- `now($type = 'datetime')`
- `extract($part, $expression, $types = [])`
- `dateAdd($expression, $value, $unit, $types = [])`
- `dayOfWeek($expression, $types = [])`

Essas funções do CakePHP, quando somadas a um conhecimento maior de SQL, aumentam o seu leque de possibilidades, então não podíamos deixar de citá-las.

8.5 Condições avançadas

Neste tópico, quero abordar um pouco mais a elasticidade que o `Query Builder` do CakePHP oferece, mais especificamente o método `where`. Até agora, nos exemplos apresentados, passamos condições simples, apenas testando se um campo tem um determinado valor, mas sabemos que há situações em que precisamos combinar uma, duas ou mais condições com operadores lógicos como `and`, `or` ou outra função qualquer.

Por exemplo, vamos supor que quero selecionar os registros da tabela `pets` que tenham `user_id` igual a `3` e `views` igual a `10`. Para construir essa consulta temos que escrever as linhas:

```
$query = $this->Pets->find();
$query->where(['user_id' => 3, 'view' => 10]);

$result = $query->first();
```

O `where()` pode receber como parâmetro um `array` com todas as condições necessárias na consulta. Essa construção é equivalente a uma consulta `SQL` usando `AND`:

```
SELECT * FROM pets WHERE user_id = 3 AND view = 10;
```

Podemos adicionar `n` condições para a `query`, fica a critério da lógica em questão e dos resultados que queremos obter. Para usar a cláusula `or` no `where()`, também passamos um *array* com as condições, assim como fizemos com `and` anteriormente. Seguindo o exemplo anterior, se quero consultar registros na tabela `pets` com `views` igual a `10` ou `7`, neste caso, teremos:

```
$query = $this->Pets->find();
$query->where([
                  'user_id' => 3,
                'OR' => [['view' => 10], ['view' => 7]]]);

$result = $query->first();
```

A condicional `or` entra como mais um item do `array` que conta os valores que serão usados para filtrar os resultados. O SQL geraria a mesma resposta:

```
SELECT * FROM pets WHERE user_id = 3 AND (view = 10 OR view = 7);
```

Você pode ter observado que o `and` não precisa ser declarado como o `or`. Implicitamente, a cada novo item adicionado no `array` contendo as condições para serem executadas no `where()`, a instrução `AND` é adicionada. O `where()` trabalha ainda como a classe `QueryExpression`, que fornece uma interface para criar consultas mais complexas dentro do `where()`, mas sem usar *arrays* na construção. Por exemplo, a última consulta que criamos com `QueryExpression` seria:

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
```

```
        return $exp
            ->eq('user_id', 3)
            ->or(function (QueryExpression $or) {
                    return $or->eq('view', 10)->eq('view', 7);
                });
    });
```

O resultado desse código é idêntico e a diferença vai ficar para a sintaxe, que deixa a construção da `query` com as cláusulas mais explícitas e evita o uso de *arrays*. É interessante destacar que usar `QueryExpression` permite usar muito mais expressões do que `or()` ou `and()`. A lista de métodos disponíveis é:

- **eq()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->eq('user_id', 3)
    });
```

```
SELECT * FROM pets WHERE user_id = 3;
```

- **notEq()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->notEq('user_id', 3)
    });
```

```
SELECT * FROM pets WHERE user_id != 3;
```

- **like()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
```

```
        return $exp->like('name', '%S')
    });
```

```
SELECT * FROM pets WHERE name LIKE "%S";
```

- **notLike()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->notLike('name', '%S')
    });
```

```
SELECT * FROM pets WHERE name NOT LIKE "%S";
```

- **in()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->in('user_id', [1, 2, 3])
    });
```

```
SELECT * FROM pets WHERE user_id IN (1, 2, 3);
```

- **notIn()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->notIn('user_id', [1, 2, 3])
    });
```

```
SELECT * FROM pets WHERE user_id NOT IN (1, 2, 3);
```

- **gt()**

```
use Cake\Database\Expression\QueryExpression;
```

```
$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->gt('views', 1000)
    });
```

```
SELECT * FROM pets WHERE views > 1000;
```

- **gte()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->gte('views', 1000)
    });
```

```
SELECT * FROM pets WHERE views >= 1000;
```

- **lt()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->lt('views', 1000)
    });
```

```
SELECT * FROM pets WHERE views < 1000;
```

- **lte()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->lt('views', 1000)
    });
```

```
SELECT * FROM pets WHERE views <= 1000;
```

- **isNull()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->isNull('breed_id')
    });
```

```
SELECT * FROM pets WHERE breed_id IS NULL;
```

- **isNotNull()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->isNotNull('breed_id')
    });
```

```
SELECT * FROM pets WHERE breed_id IS NOT NULL;
```

- **between()**

```
use Cake\Database\Expression\QueryExpression;

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp) {
        return $exp->between('views', 7, 10)
    });
```

```
SELECT * FROM pets WHERE views BETWEEN 7 AND 10;
```

- **exists()**

```
use Cake\Database\Expression\QueryExpression;

$subquery = $this->Pets->find()
    ->select(['id'])
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->equalFields('breed_id', 'breeds.id');
    });

$query = $this->Pets->find()
```

```
    ->where(function (QueryExpression $exp, Query $q) use ($subquery) {
        return $exp->exists($subquery);
    });
```

```
SELECT * FROM pets WHERE EXISTS (SELECT id FROM breeds WHERE pets.id =
breed.id)
```

- **notExists()**

```
use Cake\Database\Expression\QueryExpression;

$subquery = $this->Pets->find()
    ->select(['id'])
    ->where(function (QueryExpression $exp, Query $q) {
        return $exp->equalFields('breed_id', 'breeds.id');
    });

$query = $this->Pets->find()
    ->where(function (QueryExpression $exp, Query $q) use ($subquery) {
        return $exp->notExists($subquery);
    });
```

```
SELECT * FROM pets WHERE NOT EXISTS (SELECT id FROM breeds WHERE pets.id =
breed.id)
```

A classe `QueryExpression` é, sim, um recurso bem atraente, pois cobre um grupo considerável de cláusulas SQL e oferece uma abordagem deferente do que foi visto até aqui com o `where()`. Mas, repito, essa classe não exclui o uso do `array` para criar expressões como o `IN`, por exemplo:

```
$query = $this->Pets->find()
    ->where(['id IN' => '1,2,3']);
```

A escolha fica para o desenvolvedor ou desenvolvedora, que pode optar pela melhor, mais fácil e mais adequada forma de criar suas consultas para o projeto.

8.6 Consultas dinâmicas

Se refletirmos um pouco sobre como uma consulta é formada, para consultas mais simples, existe um padrão, que é o de geralmente adicionar um `where` com uma ou mais condições usando um operador `or` ou `and`. O CakePHP pode criar consultas dinamicamente, desde que sejam semelhantes ao que acabamos de caracterizar como uma consulta simples. Chamamos isso de *Dynamic Finders*, ou consultas dinâmicas, em português.

Vejamos um exemplo para sancionar o que estamos discutindo agora. Suponha que queremos consultar registros na tabela `users` que tenham um certo `email`. Usando o que aprendemos até aqui, faríamos:

```
$query = $this->Users->find();
$query->where(['email' => 'jozecarlos@casadocodigo.com']);
$result = $query->all();
```

Até aqui, nada novo ou complexo, mas, usando *Dynamic Finders*, todo o código escrito seria:

```
$query = $this->Users->findByEmail('jozecarlos@casadocodigo.com');
```

É isso, simples assim. O CakePHP entende que o método `findByEmail` deve procurar na tabela `users` pela coluna `email` que tenha o valor `jozecarlos@casadocodigo.com`. Toda a lógica da consulta é criada dinamicamente pelo framework.

Talvez você esteja pensando: "Por que, então, aprender; ou melhor, por que devemos criar um objeto `Query` se podemos criar uma consulta mais simples com o `where` e, depois, com o resultado, trabalhar com os dados retornados pelo banco de dados? A resposta é bem simples: filtrar ao máximo os dados com consultas no banco de dados é ainda a maneira mais rápida, em vez de fazer isso localmente no código com a sua lógica, talvez usando alguma estrutura de programação, como um *for*.

Uma das características marcantes de um framework é a versatilidade para sempre fornecer *n* meios para que a pessoa desenvolvedora consiga codificar sua lógica, mesmo que seja uma maneira mais "verbosa" como `QueryExpression`, na qual podemos utilizar funções com diversos parâmetros para criar uma lógica mais robusta. Não conseguiríamos a mesma complexidade utilizando o *Dynamic Finders*, mas o que precisamos assimilar agora é que temos esse recurso, essa carta excepcional, no CakePHP.

Recapitulando: *Dynamic Finders* são consultas geradas automaticamente a partir da junção do prefixo `findBy` ou `findAll` com o nome da coluna que se deseja usar como critério da consulta. Vejamos mais alguns exemplos:

```
$query = $this->Pets->findByName('Supino');
$query = $this->Pets->findAllByBreedId(2);
$query = $this->Users->findAllByUsername('CasaDoCodigo);
```

A diferença entre `findBy` e `findAll` é que a primeira retorna apenas um registro e a outra, uma coleção de registros, respectivamente. *Dynamic Finders* também suportam usar `and` e `or` para agregar mais de uma coluna nos critérios da consulta. Seguem alguns exemplos:

```
$query = $this->Pets->findByNameOrUserId('Supino', 3);
$query = $this->Pets->findAllByBreedIdAndUserId(2, 1);
$query = $this->Users-
>findByUsernameAndEmail('CasaDoCodigo','jozecarlos@casadocodigo.com');
```

Ou seja, para usar duas colunas como critério, utilizamos `or` ou `and` e passamos, obviamente, dois valores. Como já mencionei antes, essas são, sim, consultas simples, porém esse é um recurso bem útil e que vai ajudar você a ganhar mais velocidade e escrever menos.

Tudo isso que apresentamos é realmente bem usual, mas ainda podemos pensar: "bem, e se eu quiser criar uma consulta com mais recursos e usar um método cuja assinatura eu possa definir?". Claro que já sabemos a resposta, e o nosso amigo CakePHP fornece um

meio para criarmos nossas próprias consultas. Para criar um *Custom Finder*, precisamos criar um método na classe `Table` que represente a tabela onde a consulta deve ser realizada. Por exemplo, quero criar um método para localizar os registros na tabela `pets` que comecem sempre com a letra `S`. Para essa questão, nosso código seria:

```
use Cake\ORM\Table;

class PetsTable extends Table
{
    public function findNameStartingWith(Query $query, array $options)
    {
        $name = $options['name'];
        return $query->where(['name LIKE' => '%'.$name]);
    }
}

$this->Pets->find('findNameStartingWith', ['name' => 'S']);
```

Nesse código, criamos um método chamado `findNameStartingWith`, que recebe um objeto `Query` e um `array`. Essa assinatura é um padrão que deve ser seguido. Dentro do método, criamos a lógica necessária, segundo a qual os valores repassados para o método `where()` são representados pelo parâmetro `array $options`. No exemplo anterior, usamos o operador do SQL `LIKE` para criar a consulta, mas também poderíamos usar a classe `QueryExpression` sem problemas. Então, com uma instância de `PetsTable`, invocamos o método através do `find()`.

8.7 Associações entre tabelas

Sabemos que as entidades de um banco de dados podem se relacionar entre si, isso é uma característica comum que encontramos quando estamos modelando um sistema. As

informações, embora agrupadas por contexto, também podem ser conectadas. Por exemplo, no capítulo 3, quando criamos as *migrations* para as tabelas, vimos que a tabela `pets` estava relacionada com as tabelas `users` e `breeds` através de chaves estrangeiras, definidas por *ForeignKey*.

No banco de dados, o princípio que define o grau de relação entre tabelas é chamado de *cardinalidade* e os quatro tipos de relacionamento para os quais o CakePHP dá suporte e que vamos estudar a seguir são: `hasOne`, `belongsTo`, `hasMany` e `belongsToMany`.

hasOne

Essa associação define quando uma entidade tem uma relação única com a outra. Tomando nosso projeto de estudo AdPET como exemplo, um usuário tem um endereço, ou seja `user hasOne address`. Em outras palavras, o endereço é único para cada usuário do sistema, então é um relacionamento `1:1`. Para esse tipo de relacionamento funcionar no CakePHP, a tabela `addresses` tem que possuir uma chave estrangeira `user_id` como coluna. Toda configuração dos relacionamentos deve ser feita nas classes `Tables` do modelo e o método que declara a associação é `hasOne($associated, array $options = [])`. Segue o exemplo:

```
use Cake\ORM\Table;

class UsersTable extends Table
{
    public function initialize(array $config): void
    {
        parent::initialize($config);
        $this->hasOne('Addresses');
    }
}

Users hasOne Addresses    addresses.user_id
```

belongsTo

É também um relacionamento `1:1`. Para entender o `belongsTo`, podemos usar o mesmo exemplo de `hasOne`, mas agora podemos dizer que um endereço pertence a um usuário, `adressess belongsTo users`. É uma nova leitura do mesmo contexto. Perceba que esse relacionamento é um complemento natural do `hasOne`: para um usuário ter um endereço é porque um endereço pertence ao usuário. Esse tipo de relacionamento também precisa de uma coluna para definir a chave estrangeira na entidade que se conecta com a outra. A configuração agora deve ser feita na classe `AddressesTable` usando o método `belongsTo($associated, array $options = [])`, como demonstrado a seguir:

```
use Cake\ORM\Table;

class AddressesTable extends Table
{
    public function initialize(array $config): void
    {
        parent::initialize($config);
        $this->belongsTo('Users');
    }
}
```

Usar `hasOne` ou `belongsTo` gera uma certa confusão, pois são bem parecidos no que diz respeito a como são vistos na perspectiva do banco de dados. A diferença vai ficar mais evidente quando precisarmos acessar, através do modelo, uma informação relacionada. Um exemplo, para este mesmo contexto, seria se precisássemos pegar a informação do endereço a partir do usuário, ou então, a partir do endereço, descobrir quem é o usuário ao qual ele pertence.

```
Users     hasOne    Addresses    addresses.user_id
Addresses belongsTo Users        addresses.user_id
```

hasMany

Usamos o método `hasMany($associated, array $options = [])` quando queremos informar que uma entidade tem uma coleção de entidades relacionadas — um relacionamento `1:N`. Por exemplo, podemos dizer que uma raça tem muitos pets, `Breeds hasMany Pets`. Para esse relacionamento funcionar, citando o exemplo, a tabela `pets` possui uma chave apontando para `breeds`, `breed_id`. Na classe `Table`, a configuração seria:

```
use Cake\ORM\Table;

class BreedsTable extends Table
{
    public function initialize(array $config): void
    {
        parent::initialize($config);
        $this->hasMany('Pets');
    }
}

Breeds hasMany Pets    pets.breed_id
```

belongsToMany

Esse relacionamento acontece quando temos entidades que se associam com várias outras entidades. Podemos também dizer que é um relacionamento `many to many`, ou `M:N`. O clássico exemplo para entender esse relacionamento é comparando artigos e *tags*. Vamos utilizar esse exemplo e não algo relacionado ao projeto AdPET, pois não temos como exemplificar o `belongsToMany`. Um artigo pertence a muitas *tags*, as quais também são compartilhadas com outros artigos. O método `belongsToMany($associated, array $options = [])` é usado para definir esse tipo de relacionamento no modelo. A seguir, temos um exemplo:

```
use Cake\ORM\Table;

class ArticlesTable extends Table
{
```

```
    public function initialize(array $config): void
    {
        $this->belongsToMany('Tags');
    }
}
```

```
Articles belongsToMany Tags    articles_tags.id, articles_tags.tag_id,
articles_tags.article_id
```

Carregamento de associações

Agora que sabemos a quais relacionamentos o CakePHP dá suporte e como configurá-los devidamente nos modelos, vamos aprender a trazer junto as informações das tabelas associadas nas consultas. Por exemplo, imagine que estamos listando os registros da tabela `pets` e que cada elemento do resultado contenha, também, as informações da raça, vindas da tabela `breeds`. Nesse caso, `pets` e `breeds` se relacionam por meio de `belongsTo`, então podemos dizer que `pets belongsTo breeds`.

As associações no CakePHP são *lazy*, ou seja, quando executamos uma consulta com o método `find()`, as informações relacionadas não são incluídas no resultado, a não ser que o método `contain($associations = null, $override = false)` seja usado e transforme as consultas em *Eager Loading*. No exemplo citado, o código para executar seria:

```
$query = TableRegistry::getTableLocator()->get('Pets')->find();
$number = $query->contain(['Breeds']);
$result = $query->all();
```

Essa consulta, com o uso do `contain()`, vai incluir em cada item retornado pela consulta informações vindas da tabela `breeds`. O CakePHP utiliza o campo `breed_id` em `pets` para localizar as informações. O `contain()` também funciona como o método `get()`:

```
$pet = $this->Pets->get($id, [
    'contain' => ['Breeds','Users', 'Messages.Users'] ]
);
```

Detalhe: nesse exemplo, passamos mais de uma tabela, `Breeds`, `Users` e `Messages.Users`. Com esta última, `Messages.Users`, estamos solicitando que o CakePHP carregue as informações de `messages` e também de `users`, cujos dados estão relacionados com `messages`. Depois que toda a configuração é feita nos modelos, nas classes `Table`, usando corretamente os relacionamentos durante a construção de uma consulta, ficará muito fácil acessar informações de outras tabelas.

8.8 Os modelos do AdPET

Finalmente, temos toda a base de conhecimento necessária para criar as classes `Table` e `Entity` do projeto AdPET, além de, claro, deixar configuradas as devidas associações entre as tabelas. Para relembrar, no capítulo 3, criamos toda a estrutura das tabelas do banco de dados usando *migrations*. Até o momento, criamos as seguintes entidades do projeto:

- `pets`: salva informações dos pets cadastrados no sistema;
- `users`: armazena os usuários do sistema;
- `addresses`: endereços dos usuários;
- `breeds`: as raças dos pets;
- `states`: os estados;
- `cities`: as cidades;
- `requestAdoptions`: armazenamento dos pedidos de adoção;
- `messages`: salva as mensagens trocadas entre os usuários e os comentários.

Uma observação: não é preciso criar classes para cada tabela. Isso só é necessário quando queremos adicionar alguma configuração específica para um modelo em questão. Os *controllers* do CakePHP conseguem criar instâncias da classe `Table` que representam uma tabela do banco de dados do projeto dinamicamente, apenas pelo fato de estarem conectados ao banco. Porém, como queremos

explorar mais o conteúdo e vamos adicionar algumas configurações para os modelos, vamos iniciar nosso trabalho com a tabela `pets`, que queremos representar com as classes `Table` e `Entity`. O código das classes então seria:

```
<?php

namespace App\Model\Table;

use Cake\ORM\Table;

class PetsTable extends Table
{
    public function initialize(array $config)
    {
        parent::initialize($config);

        $this->setDisplayField('name');

        $this->addAssociations([
            'belongsTo' => [
                'Users' => ['className' => 'App\Model\Table\UsersTable'],
                'Breeds' => ['className' => 'App\Model\Table\BreedsTable']
            ],
            'hasMany' => ['Messages']
        ]);

        $this->addBehavior('Timestamp');
    }
}
```

```
<?php

namespace App\Model\Entity;

use Cake\ORM\Entity;

class Pet extends Entity
{
```

```
    protected function _getAge()
    {
        return (intval(date('Y', time() - strtotime($this-
>_properties['birthday']))) - 1970) + 1;
    }
}
```

Vale notar que, na classe `PetsTable`, as associações são configuradas com o método `addAssociations()`. Essa é uma maneira mais prática para criá-las em massa, em casos em que o modelo tenha mais de um relacionamento. No código, estamos criando um relacionamento `belongsTo` com `Users` e outro com `Breeds` e, ainda, um relacionamento `hasMany` com `Messages`. Atenção para o atributo `className`, que recebe como valor o caminho da classe `Table` correspondente, por exemplo, `App\Model\Table\UsersTable`. Na classe `Pet`, definimos um campo virtual chamado `age` com o método `_getAge()`, que vai calcular a idade do pet a partir da data de nascimento cadastrada.

Para o projeto, vamos precisar também criar as classes para `users`. Segue o código para `UsersTable` e `User`:

```
<?php
namespace App\Model\Table;


use Cake\ORM\Table;

class UsersTable extends  Table {

    public function initialize(array $config)
    {
        parent::initialize($config);

        $this->hasOne('Addresses');
        $this->addBehavior('Timestamp');
    }
}
```

```php
<?php

namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher;
use Cake\ORM\Entity;

class User extends Entity {

    // Make all fields mass assignable except for primary key field "id".
    protected $_accessible = [
        '*' => true,
        'id' => false
    ];


    protected function _setPassword($password)
    {
        if (strlen($password) > 0) {
            return (new DefaultPasswordHasher)->hash($password);
        }
    }
}
```

Na classe `User`, destaco o método `setPassword`, que reescreve o comportamento padrão do `set` para o campo `password`. O que esse método faz é receber o valor inserido pelo modelo e criptografar; assim, quando as informações forem persistidas no banco, o valor armazenado no campo `password` será criptografado, garantindo mais segurança para os dados armazenados do usuário.

Também precisamos criar as classes `RequestAdoptionsTable`, responsáveis por mapear a tabela `request_adoptions`, que vai guardar os pedidos de adoção de cada pet e relacioná-los ao usuário que fez o pedido. Para essa tabela, precisamos apenas de `Table`, não é preciso criar uma `Entity`. Segue o código:

```php
<?php
```

```
namespace App\Model\Table;

use Cake\ORM\Table;

class RequestAdoptionsTable extends Table
{
    public function initialize(array $config)
    {
        $this->addAssociations([
            'belongsTo' => [
                'Users' => ['className' => 'App\Model\Table\UsersTable'],
                'Pets' => ['className' => 'App\Model\Table\PetsTable']
            ],
            'hasMany' => ['Messages']
        ]);

        $this->addBehavior('Timestamp');
    }
}
```

Para finalizar, vamos definir as configurações da tabela `messages`. É ela que vai guardar os comentários enviados para o `pets` e, ainda, as mensagens trocadas pelos usuários enquanto pedem informações no processo de adoção. Vale lembrar que o projeto AdPET é uma ferramenta que vai possibilitar que usuários cadastrem e disponibilizem pets para adoção. O código em questão é:

```
<?php
namespace App\Model\Table;


use Cake\ORM\Table;

class MessagesTable extends Table {

    public function initialize(array $config)
    {
        $this->belongsTo('Users')
            ->setForeignKey('owner_id');
```

```
        $this->belongsTo('RequestAdoptions')
            ->setForeignKey('request_adoption_id');

        $this->addBehavior('Timestamp');
    }
}
```

Para a classe `MessagesTable`, as associações `belongsTo` usaram o método `setForeignKey`; portanto, a tabela `messages` não segue a convenção padrão do CakePHP. Para associar com a tabela `users`, deveria existir uma coluna chamada `user_id`, porém a coluna criada foi `owner_id`. Para resolver esse problema, podemos usar o método `setForeignKey` e informar para o CakePHP qual é a coluna correta.

Como mencionamos, as outras classes não são necessárias (por exemplo, para as tabelas `cities` e `states`), mas não há problema algum caso a desenvolvedora queira criar. É até um bom exercício para praticar e dominar melhor o assunto.

Conclusão

Após vermos sobre as classes `Table` e `Entity` no capítulo anterior, agora aprendemos como utilizá-las para trazer as informações do banco de dados criando consultas. Abordamos as principais maneiras oferecidas pelo CakePHP para você, profissional ou estudante, poder criar suas consultas. Nos próximos capítulos, vamos aprofundar mais os estudos sobre cada visão do framework, iniciando com os *helpers*.

Capítulo 9
Helpers

Durante o desenvolvimento, sempre há trechos de códigos que se repetem com frequência. Essas repetições devem ser evitadas e, para isso, podemos criar funções ou componentes que geram a mesma resposta e substituir as estruturas repetidas. O CakePHP possui um conjunto de *helpers*, que podemos entender como facilitadores de código. Eles podem acelerar e facilitar muito a vida de quem desenvolve, além de garantir que o código siga boas práticas de programação.

9.1 Como usar um helper?

Para o CakePHP, *helpers* são componentes que fornecem métodos públicos para a camada de apresentação. Por exemplo, se queremos escrever qual é o *charset* da página, podemos definir diretamente com uma tag HTML:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Ou usar o *helper* da pacote `HtmlHelper` :

```
echo $this->Html->charset();
```

Pelo exemplo, podemos observar que o uso de um *helper* é apenas uma chamada de uma função que vai retornar uma resposta em HTML, simples assim. Mas qual é a vantagem de usar um *helper* se podemos escrever nosso próprio HTML sem uso algum de facilitadores? O uso de um *helper* fornece mais produtividade, pois garante que o HTML gerado seja bem formatado, além de seguir boas práticas de programação e reduzir a quantidade de linhas

escritas. Essas são algumas características que justificam o fato de chamá-los de facilitadores.

Ainda não se convenceu? Vamos para mais um exemplo. Digamos que queremos carregar uma folha de estilo na página. Com o uso de um *helper*, seria:

```
echo $this->Html->css('styles');
```

E a resposta seria:

```
<link rel="stylesheet" href="/css/styles.css" />
```

Então, qual é a maneira mais fácil e rápida de escrever o código HTML sem precisar lembrar de detalhes de uso das *Tags*? Acho que, neste momento, você já tem a resposta.

Basicamente, os *helpers* vão fornecer ajuda para escrever de forma mais programática e rápida trechos de códigos que certamente vamos utilizar na criação de sistemas web, páginas, formulários etc. Usamos como exemplo dois *helpers* do grupo `HtmlHelper`, porém, temos em nossas mãos uma lista bem variada de *helpers*:

- `Cake\View\Helper\NumberHelper`;
- `Cake\View\Helper\PaginatorHelper`;
- `Cake\View\Helper\FormHelper`;
- `Cake\View\Helper\BreadcrumbsHelper`;
- `Cake\View\Helper\FlashHelper`;
- `Cake\View\Helper\HtmlHelper`;
- `Cake\View\Helper\TimeHelper`;
- `Cake\View\Helper\UrlHelper`;
- `Cake\View\Helper\RssHelper`;
- `Cake\View\Helper\SessionHelper`;
- `Cake\View\Helper\TextHelper`.

Cada classe *helper* possui funcionalidades bem distintas. Por exemplo, na criação de formulários, em que usamos tags como `Form` e `input`, temos o `FormHelper`. Não vamos usar todos os *helpers*

citados aqui, mas, no decorrer do livro, veremos quatro deles — `FormHelper`, `HtmlHelper`, `PaginatorHelper` e `FlashHelper` — para criar algumas páginas e formulários do projeto AdPET. Sendo assim, vamos apresentá-los com mais detalhes a seguir.

9.2 HtmlHelper

Enquanto desenvolvemos nossos sistemas, provavelmente temos que usar alguma imagem ou importar uma folha de estilo. O `HtmlHelper` implementa métodos para gerar as tags mais corriqueiras. Este *helper*, implementado em `Cake\View\Helper\HtmlHelper`, tem como principal objetivo criar tags HTML, como links, imagens, tabelas, carregar CSS e JavaScript, entre outros.

CSS e JavaScript

Quando for preciso importar uma folha de estilo CSS, usamos o método `Cake\View\Helper\HtmlHelper::css(mixed $path, array $options = [])` para carregar um arquivo ou vários arquivos. Por exemplo:

```
echo $this->Html->css('tabelas');
echo $this->Html->css(['formularios', 'tabelas', 'banner']);
```

Isso geraria a seguinte saída HTML:

```
<link rel="stylesheet" href="/css/tabelas.css" />

//para o uso de múltiplos arquivos

<link rel="stylesheet" href="/css/formularios.css" />
<link rel="stylesheet" href="/css/tabelas.css" />
<link rel="stylesheet" href="/css/banner.css" />
```

Se estamos observando bem, quando queremos carregar uma folha de estilo, apenas passamos o nome do arquivo sem precisar

especificar o diretório em que elas estão localizadas. Isso porque o CakePHP possui um diretório padrão, `/webroot/css`, para que o desenvolvedor ou desenvolvedora ponha as folhas de estilo — ou seja, o arquivo do exemplo, `tabelas.css`, está localizado em `webroot/css/tabelas.css`.

Para o JavaScript, a forma de carregar é similar: os arquivos devem estar na mesma raiz que o CSS, porém em um diretório chamado `js`, e o método que deve ser usado é o `Cake\View\Helper\HtmlHelper::script(mixed $url, mixed $options)`.

```
echo $this->Html->script('bootstrap');
echo $this->Html->script(['bootstrap', 'jquery', 'my_script']);
```

E a resposta seria:

```
<script src="/js/bootstrap.js"></script>

//para o uso de múltiplos arquivos

<script src="/js/bootstrap.js"></script>
<script src="/js/jquery.js"></script>
<script src="/js/my_script.js"></script>
```

Esse *helper* também pode carregar arquivos `js` dos servidores *CDN* em vez de carregar localmente. Isso é bastante utilizado para carregar bibliotecas como o JQuery; logo, em vez da localização do arquivo do próprio projeto, passamos uma URL. Por exemplo:

```
echo $this->Html->script('https://code.jquery.com/jquery-migrate-
3.1.0.min.js');

<script src="https://code.jquery.com/jquery-migrate-3.1.0.min.js">
</script>
```

CDN

CDN — do inglês, *Content Delivery Network* — é uma tecnologia para distribuir conteúdo digital através da internet. É um recurso usado para distribuir vídeos, áudios ou imagens, mas que evoluiu e hoje é usado para distribuir códigos-fonte completos ou até sites.

Imagens

Para criar imagens, temos o *helper* `Cake\View\Helper\HtmlHelper::image(string $path, array $options = [])`, que carrega os arquivos relativos ao diretório `/webroot/img/`. Este método cria uma tag `<img />` com alguns atributos, como `alt` e `id`, que podem ser declarados como um *array* de opções, caso necessário.

```
echo $this->Html->image('logo.png', ['alt' => 'AdPet Logo']);


<img src="/img/logo.png" alt="AdPet Logo" />
```

Esse *helper* é mais versátil do que parece, porque, além de carregar imagens, ele pode criar um link com imagem, ou seja, uma tag `<a/>` com uma `<img />`, caso no *array* de opções seja passada a propriedade `url` apontando para qual *controller* deve ser redirecionada a ação. Por exemplo:

```
echo $this->Html->image("logo.png", [
    "alt" => "AdPet Logo",
    'url' => ['controller' => 'Pets', 'action' => 'index']
]);
```

Nesse código, estamos carregando a imagem `logo.png`, que, quando clicada, redireciona para o *controller* `Pets` e a *action* `index`. O resultado HTML dessa execução é:

```
<a href="/pets/index">
    <img src="/img/logo.png" alt="AdPet Logo" />
</a>
```

Links

Os links são o meio para, digamos, acessar outras áreas do sistema. São representados pela tag `<a/>` e, no CakePHP, podem ser criados com o *helper* `Cake\View\Helper\HtmlHelper::link(string $title, mixed $url = null, array $options = [])`. Esse método recebe 3 parâmetros: o nome do link que vai ser exibido; a URL que vai acessar; e um *array* de opções que permite criar alguns atributos se for necessário. Por exemplo:

```
echo $this->Html->link(
    'Login',
    '/users/login',
    ['class' => 'bt_login']
);
```

Esse código resultará em uma tag `<a/>`, que vai redirecionar o fluxo do sistema para a URL `/users/login`. Aliás, só para lembrar, essa é uma URL que foi criada no arquivo `routes.php`. Além disso, adicionamos o atributo `class` com o valor `bt_login`. A saída HTML para esse exemplo será:

```
<a href="/users/login" class="bt_login">Login</a>
```

Para passar o endereço absoluto para o atributo `href`, é preciso adicionar a propriedade `_full => true` no *array* que define as opções extras do método. Refazendo o mesmo exemplo, teremos:

```
echo $this->Html->link(
    'Login',
    '/users/login',
    ['class' => 'bt_login', _full => true]
);
```

```
<a href="http://www.adetpetdomain.com/users/login"
class="bt_login">Login</a>
```

Podemos também usar o *helper* para definir diretamente o *controller* e a *action* que vão processar a requisição direcionada pelo *link*. Por exemplo, vamos dizer que, no nosso projeto AdPET, queremos acessar a *action* `view` passando um `id` que representa um registro na tabela `pets`. O que devemos fazer nesse caso será:

```
echo $this->Html->link(
    $pet->name,
    ['controller' => 'Pets','action' => 'view',1]
    ['class' => 'bt_login']
);
```

A saída correspondente será:

```
<a href="/pets/view/1" class="bt_login">Supino</a>
```

Esse exemplo está repleto de informações bem legais para relembrar o uso dos *models*. O primeiro ponto que destaco é o `$pet->name`, com o qual estamos acessando o modelo `Pet` para acessar a propriedade `name`. Assim, construiremos o texto do link que será exibido usando o valor deste atributo, que é o nome do animal salvo no banco de dados. A *action* que vai receber a requisição que tem acesso ao valor `1` e vai tratá-la de acordo com uma lógica qualquer é a `view`. E o *controller* definido é o `Pets`. Para finalizar, o número `1` como parâmetro representa o valor do `id` no banco de dados cujas informações queremos trazer. No exemplo, a coluna `name` tem o valor `Supino` visto na representação do resultado do uso do *helper*. Mais adiante, quando voltarmos para o projeto, veremos mais exemplos desse tipo de código.

Para finalizar nossa abordagem para esse *helper*, há uma propriedade bem útil que passamos também no *array* de opções, que é a chave `confirm`. Quando usamos essa chave, informamos para o CakePHP que esse link precisa de uma confirmação do usuário antes de executar a requisição, o que é recomendado para

ações que tem por objetivo deletar algum recurso no sistema. Vamos para um exemplo para aprender melhor. Imagine que queremos deletar um pet. Então, precisamos criar o link a seguir:

```
echo $this->Html->link(
    $pet->name,
    ['controller' => 'Pets','action' => 'delete',1]
    ['class' => 'bt_login', 'confirm' => 'Tem certeza que deseja excluir o registro?']
);
```

No código, estamos fazendo uma requisição para a *action* `delete`, que, obviamente, tem por objetivo deletar algo. Então, no *helper*, passamos o `id` e a propriedade `confirm`, o que vai gerar um diálogo para o usuário confirmar a ação exibindo a pergunta `Tem certeza que deseja excluir o registro?`. Basicamente, quando o CakePHP interpreta essa instrução, ele insere um JavaScript no link para criar esse diálogo de confirmação. Veja como é a saída desse exemplo:

```
<a href="/pets/delete/1"
    onclick="return confirm(
        'Tem certeza que deseja excluir o registro?'
    );">
    Delete
</a>
```

No código gerado, é usado o evento `onclick` para chamar uma função js `confirm`. Caso o usuário confirme, a requisição vai chegar à *action* `delete`.

Tabelas

Com o *HtmlHelper*, também conseguimos criar tabelas com os seguintes métodos:

- `Cake\View\Helper\HtmlHelper::tableHeaders(array $names, array $trOptions = null, array $thOptions = null)`, usado para criar o cabeçalho da tabela;

- `Cake\View\Helper\HtmlHelper::tableCells(array $data, array $oddTrOptions = null, array $evenTrOptions = null, $useCount = false, $continueOddEven = true)` , para criar as linhas da tabela.

Vamos ver um exemplo para entender melhor o uso dos métodos. Digamos que queremos criar a seguinte tabela HTML:

```
<table>
  <tr>
    <th>ID</th>
    <th>Nome</th>
  </tr>
  <tr>
    <td>1</td>
    <td>Supino</td>
  </tr>
</table>
```

É uma tabela simples, com duas colunas, `ID` e `Nome` , com os valores `1` e `Supino` . Reescrevendo essa tabela como *helper*, teríamos:

```
<?php
<table>
  echo $this->Html->tableHeaders(['ID', 'Nome']);
  echo $this->Html->tableCells([['1', 'Supino']]);
</table>
```

Esse exemplo mostra como uma estrutura *HTML* da página pode ser criada a partir de métodos *PHP*, os quais podem ser manipulados a partir de uma regra de negócio da aplicação. Logo, temos um aumento da flexibilidade de criar código mais dinâmico com o CakePHP e escrevendo menos.

9.3 FormHelper

O `FormHelper` trabalha exclusivamente com as tags que são utilizadas em formulários, seja para criar, editar ou excluir informação. Este *helper* facilita a criação do *schema* de dados que será enviado para o *controller*, o qual pode ser acessado usando `request->getData()`, caso os dados sejam enviados com um `POST`, por exemplo.

Para iniciar um formulário, temos que chamar o método `Cake\View\Helper\FormHelper::create(mixed $context = null, array $options = [])`, que é responsável por iniciar com a tag `<form />`, e finalizar com o método `Cake\View\Helper\FormHelper::end($secureAttributes = [])`. Por padrão, quando criamos um `Form` sem passar nenhuma configuração, os dados vão ser submetidos para o *controller* atual e para a *action* `add`.

Vamos exemplificar. Assumindo que, quando navegamos, acessamos uma *action* do `PetsController`, temos:

```
echo $this->Form->create();

<!-- Campos do formulário aqui... -->

echo $this->Form->end();
```

O código vai gerar o HTML seguinte:

```
<form method="post" action="/pets/add">

<!-- Campos do formulário aqui... -->

</form>
```

Por padrão, o método usado no formulário é o *POST*, mas ele pode ser modificado se for passado um *array* de opções com a propriedade `type`, cujos valores podem ser:

- `post`: define o método do formulário para `POST`;
- `delete`: define o método do formulário para `DELETE`;
- `file`: o método do formulário é o POST, mas *enctype* é definido para `multipart / form-data`;

- `get` : define o método do formulário como `GET` ;
- `patch` : define o método do formulário para `PATCH` ;
- `put` : define o método do formulário para `PUT` .

Se queremos usar *PUT* e não *POST*, o que precisamos fazer é usar a configuração como no código a seguir:

```
echo $this->Form->create(null, ['type' => 'put');

<!-- Campos do formulário aqui... -->

echo $this->Form->end();
```

Nesse mesmo *array*, podemos redefinir qual *controller* e qual *action* vão processar a requisição.

```
echo $this->Form->create(null, [
  'type' => 'put', 'url => [
  'controller' => 'users'
  'action' => 'save']);

<!-- Campos do formulário aqui... -->

echo $this->Form->end();
```

Isso vai gerar o seguinte código HTML com as opções assinaladas:

```
<form method="put" action="/users/save">
```

Agora que entendemos o básico, precisamos também aprender como criar os campos dos formulários, ou não vamos conseguir salvar nenhum dado, não é verdade? Um formulário precisa tratar diversos tipos de entradas de dados, desde simples textos a tipos mais complexos, como listas, datas ou senhas. Dependendo do tipo de informação que o campo do formulário vai receber, temos que usar um método específico do `FormHelper` que vai gerar a tag HTML adequada para a informação. Entre os métodos disponíveis que temos para criar os campos, podemos destacar:

- `text(string $fieldName, array $options)` : cria tags do tipo *input* para inserir texto de tamanho pequeno;
- `password(string $fieldName, array $options)` : cria *input* do tipo *password* que permite inserir senhas de forma visualmente segura;
- `hidden(string $fieldName, array $options)` : cria um campo oculto;
- `textarea(string $fieldName, array $options)` : cria uma tag *textarea* para criar um campo de texto de tamanho extenso;
- `select(string $fieldName, array $options, array $attributes)` : cria uma lista de opções;
- `file(string $fieldName, array $options)` : campos que permitem selecionar e enviar arquivos;
- `checkbox(string $fieldName, array $options)` : cria um botão *checkbox*;
- `radio(string $fieldName, array $options, array $attributes)` : cria um botão do *radio*;
- `submit(string $caption, array $options)` : cria um botão para submeter os dados inseridos no formulário.

Todos esses métodos precisam receber no argumento `$fieldName` o nome da propriedade do modelo que representa a informação no formulário. Por exemplo, se quisermos criar um *form* para salvar o nome do *pet*, usaremos:

```
echo $this->Form->create();
echo $this->Form->text('name');
echo $this->Form->end();
```

Temos `name`, que é uma das propriedades de uma `Entity`, ou seja, se estamos salvando informações na tabela *pets*, logo, como aprendemos no capítulo 7, temos uma `Entity` chamada `Pet`, que possui uma propriedade chamada `name`. Na criação do campo, podemos passar uma lista de configurações específicas no segundo argumento `array $options`. Nesse campo, podemos assinalar uma classe CSS, passar um `id` ou definir um valor para o campo.

```
echo $this->Form->create();
echo $this->Form->text('name', ['class' => 'bt bt-sm']);
echo $this->Form->end();
```

Com esse exemplo, o código do formulário seria:

```
<form method="post" action="/pets/add">
  <input name="name" type="text" class="bt bt-sm">
</form>
```

Tudo o que for configuração extra do campo, como *placeholders*, `ids` etc., podemos passar em `$options`. Vamos enriquecer o aprendizado com mais detalhes criando um formulário mais completo para cadastrar um *pet* no próximo capítulo, em que vamos reusar este *form* e finalizar o cadastro. Mas, por enquanto, vamos usá-lo apenas para detalhar mais o `FormHelper`. Segue o código:

```
<?php

     echo $this->Form->create();
     echo    $this->Form->text('name',['placeholder' => 'Nome','label' =>
false]);
     echo    $this->Form->textarea('description', ['placeholder' => 'Fale
um pouco do Pet','label' => false]);
     echo    $this->Form->file('profile_picture', ['placeholder' => 'Foto
do Pet','label' => false]);
     echo    $this->Form->select('gender', ['M' => 'Macho', 'F' =>
'Fêmea'], ['placeholder' => 'Sexo','label' => false]);
     echo    $this->Form->submit('Finalizar Cadastro',['class' =>'button
large']);
     echo $this->Form->end();
?>
```

Esse código cria um formulário que vai enviar para o `PetsController`, na *action* `add`, os valores para `name`, `description`, `profile_picture` e `gender`. No formulário, quero destacar duas propriedades: a primeira é `profile_picture`, que vai representar a imagem do pet e usa o método `file()` para criar um campo que permite enviar arquivos; a segunda é o campo `gender`, que representa o sexo e usa o `select()`

para criar uma lista com os valores `Macho` e `Fêmea`. No caso do `select()`, para criar um lista de `<option>` no *HTML*, é preciso passar um *array* cuja `key` vai ser o valor do `<option>` e o `value` vai ser a opção apresentada para o usuário. A seguir, podemos observar como ficou o formulário depois de transformado em HTML.

```
<form method="post" action="/pet/add">
 <input type="text" name="name" placeholder="Nome" required="required">
<textarea name="description" placeholder="Fale um pouco do Pet" rows="5">
</textarea>
<input type="file" name="profile_picture" placeholder="Foto do Pet">
<select name="gender" placeholder="Sexo" required="required">
    <option value="M">Macho</option>
    <option value="F">Fêmea</option>
</select>
<div class="submit">
    <input type="submit" class="button large" value="Finalizar Cadastro">
</div>
</form>
```

Nos próximos capítulos, vamos criar exemplos mais completos, partindo do *controller* até a visão que utilizará tudo o que aprendemos sobre o `FormHelper`.

9.4 PaginatorHelper

O `PaginatorHelper`, `Cake\View\Helper\PaginatorHelper(View $view, array $config = [])`, tem como função criar os controles de paginação para uma determinada lista — como os botões de próxima página, anterior, página 1, 2, 3 etc. Para isso, este *helper* oferece métodos como:

- `numbers($options = [])`: cria botões que navegam para um número específico de página;
- `prev($title = '<< Previous', $options = [])`: cria um botão que navega para a página anterior;

- `next($title = 'Next >>', $options = [])` : cria um botão que navega para a próxima página;
- `first($first = '<< first', $options = [])` : cria um botão que navega para a primeira página da lista;
- `last($first = '<< last', $options = [])` : cria um botão que navega para a última página da lista;
- `counter($options = [])` : retorna a quantidade de páginas da lista.

Por exemplo, vamos supor que temos uma lista de pets e queremos paginar para que essa lista mostre *n* resultados por vez, e não todos os registros armazenados no banco de dados de uma vez na página. Poderíamos escrever o controle de paginação da seguinte forma:

```
<?php
    echo $this->Paginator->prev('« Anterior');
    echo $this->Paginator->numbers();
    echo $this->Paginator->next('Próximo »');
```

Esses controles são apenas para criar um botão para avançar ou retornar a página e criar botões para ir direto para uma página *n* (1, 2, 3 etc.). Esse bloco de código gerado em HTML seria:

```
<ul class="page-buttons">
    <li class="prev disabled button">
       <a aria-label="Previous"><< Anterior</a>
    </li>
    <li class="active button">
       <a href="#">1</a></li>
    <li>
      <a class="button" href="/?page=2">2</a>
    </li>
    <li>
      <a class="button" aria-label="Next" href="/?page=2">
         Próximo >>
       </a>
    </li>
</ul>
```

Como podemos observar, os controles de paginação são listas cujas opções são links que redirecionam para a posição indicada na propriedade `href`. Não vamos exemplificar como o *controller* vai tratar as requisições desses links, mas quero destacar como usar o *helper*. Mais adiante, vamos trabalhar na página inicial do AdPET; lá, vamos voltar nesse assunto, abordando-o com mais detalhes.

Vale ressaltar que, ao paginar seus resultados, além de proporcionar uma experiência melhor para o usuário — pois evita que todos os dados de uma tabela sejam exibidos (o que certamente será um problema se a quantidade de dados for grande) —, temos também uma melhora na performance do processamento da requisição, já que evita que o banco de dados e o servidor tenham que trabalhar com toda a massa de dados da tabela em qualquer requisição.

Tudo que abordamos sobre o `PaginatorHelper` foi para a camada da visão; porém, para que a paginação funcione, o *controller* precisa atuar nesse processo, pois vai receber as requisições dos botões do bloco de código da paginação. Esse *helper* trabalha em conjunto com o componente `PaginatorComponent`, que atua no lado do *controller* e precisa ser inicializado e configurado, como demonstrado a seguir:

```
<?php
    class PetsController extends AppController {

        public $paginate = [
            'limit' => 9
        ];

        public function initialize()
        {
            parent::initialize();
            $this->loadComponent('Paginator');
        }
       ...
}
```

A variável `$paginate` define a configuração básica para o funcionamento da paginação do lado do *controller*, que é um `array` em que podemos definir propriedades como `limit`, `sort`, `direction`, `order` e `fields`. Por exemplo, imagine que queremos paginar os resultados da tabela `pets` com nove itens por resultado, ordenar os registros pelo nome e de forma ascendente, e fazer com que os registros retornados pela paginação apenas informem o nome do pet e a data de criação. Para escrever essa configuração com o `PaginateComponent`, você pode fazer da seguinte forma:

```
<?php
    class PetsController extends AppController {

        public $paginate = [
         'fields' => ['Pets.name', 'Pets.created'],
         'limit' => 9,
         'order' => [
             'Pets.name' => 'asc'
         ]
    ];

    public function initialize()
    {
        parent::initialize();
        $this->loadComponent('Paginator');
    }
        ...
}
```

Vale ressaltar que `fields` e `order` são propriedades que, para funcionar, devem receber como valor outro `array` definindo que coluna queremos ordenar de forma ascendente (com o `asc`) ou descendente (com o `desc`).

Ainda no *controller* que vai receber requisições do `PaginatorHelper`, precisamos iniciar o componente usando `$this->loadComponent('Paginator')`. O template HTML gerado para a

paginação também pode ser modificado como configuração; porém, isso deve ser feito na classe `AppView` :

```
 <?php
    class AppView extends View
{

    public function initialize()
    {
        $this->loadHelper('Paginator', ['templates' => 'paginator-
templates']);
    }
}
```

A propriedade `template` recebe como valor `paginator-templates` , que é o nome do arquivo que será usado como template para o `PaginatorHelper` . Esse arquivo deve ser criado em `/config/paginator-template.php` e pode definir mudanças no template se utilizar as propriedades:

- `nextActive` : template do botão gerado pelo método `next()` com estado do link ativo;
- `nextDisabled` : template do botão gerado pelo método `next()` com estado do link desativado;
- `prevActive` : template do botão gerado pelo método `prev()` com estado do link ativo;
- `prevDisabled` : template do botão gerado pelo método `prev()` com estado do link desativado;
- `counterRange` : o template para o retorno do `counter()` quando `format == range` ;
- `counterPages` : o template para o retorno do `counter()` quando `format == pages` ;
- `first` : template usado pelo método `first()` para gerar o botão que volta para a primeira página;
- `last` : template usado pelo método `last()` para gerar o botão que navega para a última página;

- `number` : template usado pelo método `number()` para gerar os botões dos números das páginas;
- `current` : template usado para exibir a página atual;
- `ellipsis` : template usado para exibir *elipses* no retorno do método `numbers()` para informar que existem mais páginas para serem exibidas;
- `sort` : template usado para o `sort()` , mas que não ordena de forma ascendente ou descendente;
- `sortAsc` : template usado para o `sort()` para ordenar de forma ascendente;
- `sortDesc` : template usado para o `sort()` para ordenar de forma descendente.

Vamos a um exemplo prático para ajudar a compreender como podemos usar essas propriedades, que, na verdade, são templates para cada botão gerado para a paginação. Imagine que queremos mudar o template dos botões `prev()` e `next()` para que exibam um ícone do famoso `FontAwesome` . Tudo o que precisamos fazer é:

```
  <?php
    $config = [
        'nextActive' => '<li><a class="button" href="{{url}}"><i class="fa
fa-caret-right"></i></a></li>',
        'nextDisabled' => '<li class="next disabled button"><a><span><i
class="fa fa-caret-right"></i></span></a></li>',
        'prevActive' => '<li><a class="button" href="{{url}}"><i class="fa
fa-caret-left"></i></a></li>',
        'prevDisabled' => '<li class="prev disabled button"><a><span><i
class="fa fa-caret-left"></i></a></li>'
    ];
```

Nesse exemplo, redefinimos o template de `nextActive` , `nextDisabled` , `prevActive` e `prevDisabled` e adicionamos um ícone, `<i class="fa fa-caret-left"></i>` , dentro da tag `<li><a></a></li>` . Essa configuração para o `PaginatorHelper` vai gerar novos botões dentro da lista `<ul/>` , criando uma lista de botões para o controle da paginação.

9.5 FlashHelper

A função do `FlashHelper`, da classe `Cake\View\Helper\FlashHelper(View $view, array $config = [])`, é notificar visualmente o usuário com alguma mensagem, seja ela para informar quando algum problema ocorre ou para informar o sucesso de alguma operação.

Assim como o `PaginatorHelper`, este *helper* também trabalha em conjunto com um componente no lado do *controller*, o `FlashComponent`, pois a responsabilidade de repassar informação para o `FlashHelper` é do *controller*. Na *view*, usar esse *helper* é bem simples:

```
<?= $this->Flash->render() ?>
```

A mensagem que aparece na página vem de uma variável de sessão que, por padrão, é `$_SESSION`. Seu valor pode ser definido no controller com o método `$this->Flash->set()`.

```
<?= $this->Flash->set('Casa do Código'); ?>
```

Esse componente também trabalha com templates, justamente para podermos controlar como as mensagens serão exibidas. Os templates para `FlashHelper` devem ser criados em `templates/element/flash`, onde serão criados arquivos `.php` para representar diversos tipos de mensagens — por exemplo, quando queremos enviar mensagem de erro para usuários. Nesse caso, podemos criar um template chamado `error.php` e definir dentro o template da mensagem:

```
<?php
if (!isset($params['escape']) || $params['escape'] !== false) {
    $message = h($message);
}
?>
<div class="message error" onclick="this.classList.add('hidden');"><?=
$message ?></div>
```

Então, no *controller*, podemos passar a mensagem usando:

```
$this->Flash->set('Um error ocorreu', [
    'element' => 'error'
]);
```

Ou:

```
$this->Flash->error('Um error ocorreu');
```

Esse último método é uma forma mais abreviada para enviar uma mensagem já informando qual template usar na *view*. Caso tenhamos um template, digamos, `templates/element/flash/success.php`, podemos chamar o método `$this->Flash->success('Sucesso!')`.

9.6 Carregando um helper

Os *helpers* que apresentamos neste capítulo são autocarregados pelo framework. Isso quer dizer que não precisamos deixar explícito o seu uso. Por exemplo, o CakePHP vai automaticamente deixar o `FormHelper` disponível para uso nas *views* sem que o programador precise escrever alguma configuração.

Porém, existem situações em que precisamos carregar alguma classe de um plugin externo ou queremos mesmo explicitar o que estamos carregando, redefinindo alguma configuração. Nesses casos, um *helper* deve ser carregado na classe `AppView` localizada em `src/View/AppView.php` com o método `loadHelper($name, array $config = [])`. Segue o exemplo:

```
class AppView extends View
{
    public function initialize()
    {
        parent::initialize();
        $this->loadHelper('Html');
        $this->loadHelper('Form');
        $this->loadHelper('Flash');
```

```
        $this->loadHelper('Picture.Upload');
        $this->loadHelper('Paginator',
                          ['templates' => 'paginator-templates']);
    }
}
```

O uso do método `loadHelper()` é bem simples: precisamos apenas informar o nome do *helper* que precisa ser inicializado e, caso precise passar alguma configuração para mudar algum comportamento, o método pode receber um segundo argumento `array $config = []` com as configurações, como observado no `$this->loadHelper('Paginator', ['templates' => 'paginator-templates'])`. Nesse caso, estamos informando para `PaginatorHelper` que o template a ser exibido está no arquivo `paginator-templates`. No código `$this->loadHelper('Picture.Upload');`, estamos informando que precisamos carregar um *helper* do plugin `Picture` chamado `Upload`. Lógico que isso funcionará somente se realmente tivermos um plugin instalado no projeto com essa nomenclatura.

Conclusão

Este foi um capítulo âncora para os próximos, nos quais vamos aprofundar mais nosso conhecimento sobre *views* e vamos, de fato, codificar as páginas, entrar em detalhes na camada de apresentação e criar formulários. Dentro desse contexto, vamos poder ver com mais detalhes o funcionamento de alguns *helpers* que abordamos aqui.

Capítulo 10
Views

Segundo o *pattern* MVC, a camada de visualização tem como responsabilidade apresentar as informações que vêm desde a camada de dados e são repassadas pelos *controllers*. É nessa camada que ocorre todo o tratamento de como as informações serão exibidas, como aplicação de estilos, formatação de textos, ou apenas a disponibilização um download de um arquivo PDF, por exemplo. A seguir, vamos estudar o que o CakePHP oferece de recursos para essa camada.

10.1 Templates

O CakePHP é bem versátil. Com ele, podemos gerar páginas HTML, bem como usar XML ou JSON para apresentação de dados, que são tipos de documento bem comuns quando estamos desenvolvendo com web service ou uma API RESTful.

No sistema AdPET, queremos que as informações sejam exibidas para os usuários através de páginas HTML carregadas no browser; logo, tudo o que precisamos é criar arquivos `.php` para representar os templates.

As *views* recebem informações diretamente dos *controllers*, enquanto os templates, uma combinação de tags HTML e PHP, vão produzir uma resposta no formato HTML legível para os *browsers*. Para trabalhar com templates, também devemos seguir uma convenção a fim de facilitar o desenvolvimento da aplicação. Os arquivos devem ser criados dentro do diretório `/templates/` e os nomes dos arquivos devem ser os mesmos das *actions* criadas nos *controllers*.

Calma, vou exemplificar para facilitar o entendimento! Esse fluxo que a informação percorre entre as camadas deve ser bem compreendido para aprendermos exatamente como trabalhar com o CakePHP. Digamos que queremos listar todos os *pets* cadastrados no sistema e o arquivo o qual vai gerar essa lista é o `index.php`, localizado no diretório `templates/Pets/index.php`.

Já sabemos que temos uma tabela no banco de dados chamada *pets*. Para ler as informações dessa tabela, vamos precisar de uma classe chamada `PetTable`, que representa o modelo que é acessado por um *controller* chamado `PetsController`. Como queremos que o arquivo `index.php` apresente algum conteúdo, precisamos que uma *action* nomeada `index()` repasse as informações.

Simplificando, toda vez que é preciso apresentar informações em uma página, temos que criar um template que deve ter o mesmo nome de uma *action* no *controller*. Essa é a convenção natural de comunicação entre as camadas. Se, por algum motivo, o nome do arquivo precisar ser diferente da *action*, é preciso informar o nome do arquivo a ser utilizado.

10.2 Definindo variáveis para a view

Nesse momento, você já deve ter percebido que tudo no CakePHP é bem simples, e disponibilizar alguma informação para a *view* não foge a essa regra. Tudo o que precisamos saber é que temos que usar o método `set()`, definido em `Cake\Controller\Controller::set(string $var, mixed $value)`, para criar uma variável que exponha um determinado dado para o template. Por exemplo:

```
class PetsController extends Controller
{
  index(){
    $table_pets = TableRegistry::get('Pets');
    $query = $table_pets->find('all');
    $pets = $query->all();

    $this->set('pets', $pets);
  }
}
```

A *action* `index()` lista todos os dados da tabela `pets` e os armazena na variável `$pets`, a qual foi definida pelo método `set()`. Neste momento, essa variável fica acessível para o template `index.php`, onde podemos utilizar alguma lógica e apresentar as informações na página como queremos. Para fixar melhor esse conteúdo, vamos usar o projeto AdPET e criar nossa página inicial. Usaremos o bloco de código a seguir no arquivo `index.php`:

```
<div class="row no-collapse-1">

    <?php foreach ($pets as $pet): ?>
        <section class="4u pet_block">
            <?= $this->Html->image("/uploads/".h($pet->photo_name), [
                "alt" => h($pet->name),
                "class" => "image featured",
                'url' => ['action' => 'view',  $pet->id]
            ]); ?>
            <div class="views">Visualizações: <?= $pet->views ?></div>
            <div class="box">
                <h3><strong><?= $this->Html->link( h($pet->name),
['action' => 'view', $pet->id]) ?></strong></h3>
                <p><?= h($pet->description) ?></p>
                <?= $this->Html->link('Adote','/adopt',['class' =>
'button']) ?> </div>
        </section>
    <?php endforeach; ?>
```

```
</div>
```

Observando esse código, é possível manipular as informações contidas na variável `$pets` usando a estrutura de controle `foreach` para ler cada elemento e montar a estrutura final em *HTML*. Note que estamos lidando com um `array` de objetos da classe `Pet`, que é uma `Entity`, resultado do método `$table_pets->find('all')`.

Na interação do laço, acessamos os atributos da classe `Pet` e usamos para montar a estrutura da página com as informações obtidas ao acessar propriedades do objeto, como em `$pet->name`. Existem muitos detalhes no código que ainda vamos discutir, mas, já adiantando e matando um pouco a curiosidade, nossa página inicial será assim:

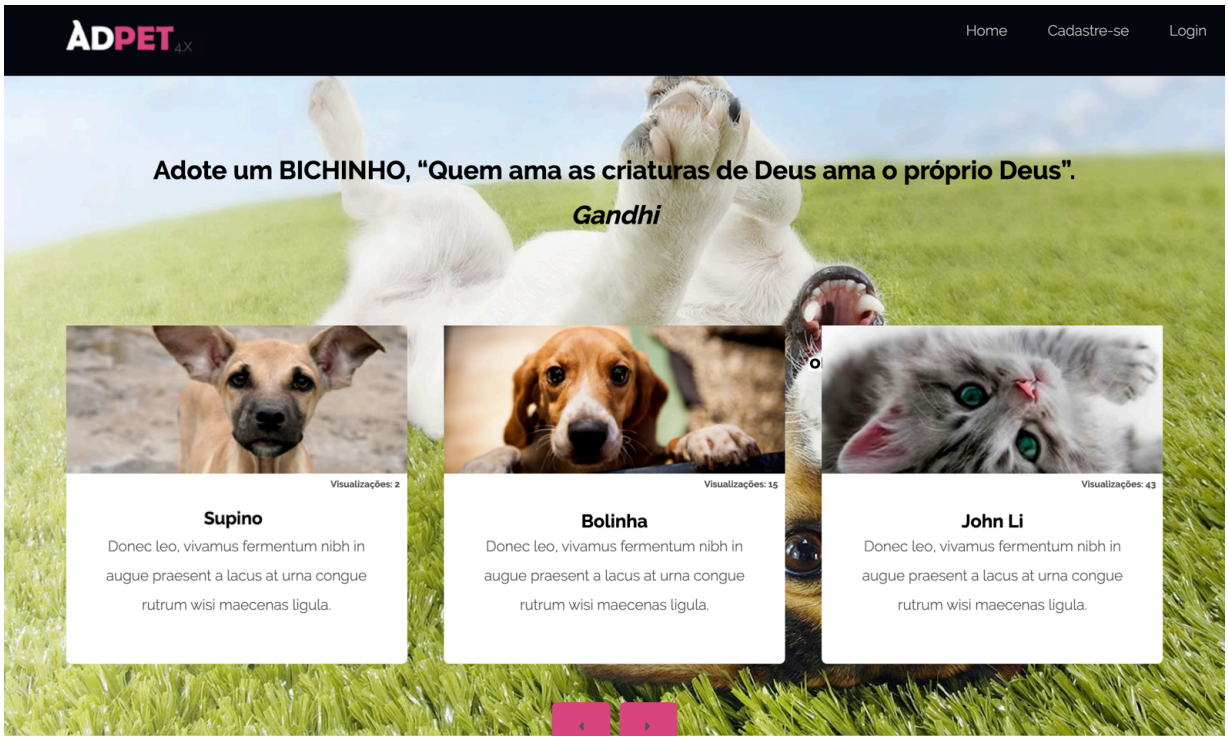


Figura 10.1: Página inicial do sistema AdPET.

10.3 Layouts

No CakePHP, existem templates especiais que funcionam como um *container* (contêiner), ou seja, são templates que têm como função ser a base de código comum entre as páginas. O leiaute padrão de uma aplicação CakePHP fica localizado em `templates/layout/default.php`. Nesse arquivo, podemos adicionar elementos como arquivos CSS, JavaScript e toda a estrutura comum entre as páginas que serão exibidas, como banners, o menu principal etc.

Na figura anterior, onde mostramos como ficou o arquivo `index.php`, o que apresentamos é, na verdade, o resultado final da combinação deste arquivo com o arquivo `default.php`, que usamos para criar o esqueleto da aplicação AdPET. Vamos dar uma olhada em como esse arquivo foi codificado — temos coisas interessantes para comentar:

```
<?= echo $this->Html->docType(); ?>
<html>
<head>
    <?= $this->Html->charset() ?>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><?= $this->fetch('title') ?></title>
    <?= $this->Html->meta('icon') ?>
    <!--[if lte IE 8]><?= $this->Html->script('ie/html5shiv') ?><![endif]-
->
    <?= $this->Html->script(['jquery.min', 'jquery.dropotron',
'skel.min','skel-layers.min','init']) ?>
    <?= $this->Html->css(['skel', 'style', 'style-wide']) ?>
    <!--[if lte IE 8]><?= $this->Html->css('ie/v8') ?><![endif]-->
</head>

<body>
    <!-- Wrapper -->
    <div class="wrapper style1">
        <!-- Header -->
        <?php echo $this->element('topbar'); ?>
        <!-- Banner -->
```

```
        <?php echo $this->element('banner'); ?>
        <!-- Extra -->
        <div id="extra">
            <div class="container">
                <?= $this->fetch('content'); ?>
            </div>
        </div>
    </div>
    <!-- Footer -->
    <?php echo $this->element('footer'); ?>
    <!-- Copyright -->
    <?php echo $this->element('copyright'); ?>
</body>
</html>
```

Lembra quando comentei que o leiaute funciona como um *container* das páginas da aplicação e que os elementos definidos nele são repetidos nas diversas *actions* que exibem suas *views*? Esse comportamento é criado pelo método `$this->fetch('content')`, que é responsável por incluir o resultado de uma *action* dentro do *layout*. No exemplo do `index.php`, a exibição desse arquivo vai ser criada dentro de `<div class="container">`. Isso porque essa *tag* está envolvendo tudo que for retornado por `$this->fetch('content')`.

Uma boa prática de programação é quebrar a página em pequenos blocos de código que podem ser reutilizados. Isso vai facilitar a manutenção e melhorar a organização e legibilidade do código. No exemplo dado anteriormente, para criar pequenos blocos de códigos, foi usado um *element*, que é uma funcionalidade da classe `Cake\View\View::element(string $elementPath, array $data, array $options = [])`. No código do arquivo `default.ctp` descrito anteriormente, temos um *element* para definir o leiaute do topo da página no trecho de código `$this->element('topbar')`.

Por padrão, quando requisitamos uma *action*, ela vai usar imediatamente o `default.php`; porém, dependendo do contexto, às vezes precisamos exibir outro tipo de estrutura em determinadas páginas, um novo leiaute. Mudar o arquivo do leiaute que vai ser

exibido em uma *action* pode ser feito invocando o método `$this->viewBuilder()->setLayout(string $name)`, onde `$name` é o nome do arquivo que queremos usar. Vale lembrar que os novos leiautes também devem ser mantidos no diretório `templates/layout/`.

Por exemplo, supondo que temos *views* que representam páginas administrativas, podemos criar um leiaute chamado `admin.php` e utilizá-lo em uma *action* desta maneira:

```
class PetsController extends Controller
{
  index(){

    $this->viewBuilder()->setLayout('admin');
    ...
  }
}
```

É um recurso que pode ser bem útil no projeto, pois é bem comum uma aplicação web mudar a sua estrutura de leiaute de acordo com o perfil do usuário que a está utilizando.

10.4 Cabeçalho do leiaute

No elemento `head` do arquivo `default.php`, devem ser importados todos os arquivos CSS e JavaScript que fazem parte do projeto e definem informações como títulos, ícones etc. Os arquivos foram importados usando o `HtmlHelper`, *helper* que estudamos no capítulo anterior. Essa classe contém métodos que facilitam a criação de blocos de código, ou a importação de arquivos externos, como folhas de estilo CSS ou arquivos JavaScript, entre outros. No código do leiaute exibido na seção anterior, vimos os seguintes métodos:

```
<?= $this->Html->meta('icon') ?>
<?= $this->Html->script(['jquery.min', 'jquery.dropotron',
```

```
'skel.min','skel-layers.min','init']) ?>
<?= $this->Html->css(['skel', 'style', 'style-wide']) ?>
```

Os métodos `meta()`, `script()` e `css()` são *helpers* que adicionam metainformações na página e importam arquivos JavaScript e CSS, respectivamente. Quando a página for exibida, seguindo o exemplo, a saída HTML correspondente será:

```
<link href="/favicon.ico" type="image/x-icon" rel="icon"/>
<link href="/favicon.ico" type="image/x-icon" rel="shortcut icon"/>
<script src="/js/jquery.min.js"></script>
<script src="/js/jquery.dropotron.js"></script>
<script src="/js/skel.min.js"></script>
<script src="/js/skel-layers.min.js"></script>
<script src="/js/init.js"></script>
<link rel="stylesheet" href="/css/skel.css"/>
<link rel="stylesheet" href="/css/style.css"/>
<link rel="stylesheet" href="/css/style-wide.css"/>
```

No CakePHP, em vez de escrevermos código HTML diretamente no template, é uma boa escolha delegar essa responsabilidade para um *helper*, pois mantém o código mais simples e evita possíveis erros ao escrever os elementos.

10.5 Elements

Na seção que tratamos sobre o leiaute, foi apresentado o *element*. Para o CakePHP, *element* é todo e qualquer bloco de código que podemos usar em diversas páginas. Um *element* deve ser criado no diretório `templates/element/` e podemos criar quantos elementos forem necessários para deixar o código mais reutilizável. No leiaute da aplicação, os elementos criados foram:

```
<?php echo $this->element('topbar'); ?>
<?php echo $this->element('banner'); ?>
```

```
<?php echo $this->element('footer'); ?>
<?php echo $this->element('copyright'); ?>
```

Por exemplo, o código do *element* `banner`, criado em `templates/element/banner.php`, é:

```
<div id="banner" class="container">
    <section>
       <?= $this->Flash->render() ?>
       <?= $this->Flash->render('auth') ?>
    </section>
</div>
```

Esse bloco de código ficou simples e pequeno porque usamos o *element* para dividi-lo. Note que usamos o `FlashHelper`, *helper* para imprimir algum tipo de mensagem. A linha escrita, `$this->Flash->render('auth')`, vai imprimir uma mensagem do sistema de autenticação do CakePHP — tópico que será abordado no capítulo 12.

O método `$this->element('banner')` importa o arquivo `templates/element/banner.php` e insere o seu conteúdo no `default.php`. Uma pequena peculiaridade sobre o método `$this->element` é que ele também permite passar informação para dentro do template usando um *array* de dados. Por exemplo, se quisermos imprimir a versão do CakePHP que estamos utilizando neste livro, poderíamos passar como parâmetro `$version` para o *element* `toolbar.php`. Apenas precisamos passar esse valor da maneira descrita a seguir:

```
echo $this->element('toolbar', [
    "version" => "4.X"
]);
```

Os *elements* também podem criar uma cache caso a pessoa desenvolvedora deseje. Podemos ter estruturas que sequer vão precisar exibir alguma informação dinâmica, apenas vão exibir dados estáticos. Habilitar cache para estes casos em que as informações não vão mudar vai melhorar a performance da aplicação. O código a seguir exemplifica o uso de cache:

```
echo $this->element('toolbar',
  ["version" => "4.X"],
  ['cache' => true]
);
```

Tenha em mente que *elements* são *views* que não devem possuir regras de negócio, devem ser simples. Para blocos de códigos mais complexos, que acessam camada de modelo e contêm regras de negócio, o CakePHP disponibiliza as `Cell`, assunto do próximo tópico.

10.6 Cell

Dentro de uma página, podem existir estruturas que pretendemos reutilizar (motivação já apresentada no tópico anterior), mas que, ainda que sejam pequenos blocos de códigos, precisam ter um comportamento único, implementando uma regra de negócio. No CakePHP, `Cell` é um tipo de componente que possui um template `ctp` e tem uma classe que controla o seu comportamento. Tem certa semelhança com `Element`, mas este apenas precisa de um template para funcionar.

Vamos ao exemplo prático. Para criar uma `Cell`, podemos utilizar o comando `bake`. É uma forma mais rápida para gerar os arquivos nos lugares apropriados:

```
bin/cake bake cell ViewPets
```

Esse comando vai gerar uma classe em `src/View/Cell`, lugar padrão para este componente do CakePHP e é onde você deve escrever a lógica necessária. Segundo o resultado gerado pelo comando, a classe gerada será `src/View/Cell/ViewPetsCell`. Vamos aproveitar esse arquivo e implementá-lo para utilizar no projeto AdPET. Esta `Cell` vai ter a função de exibir mais pets da mesma raça, o que vai acontecer quando, na tela inicial, o usuário clicar em um bichinho

para ver os detalhes — onde teremos uma seção para exibir mais bichinhos como sugestão, filtrando apenas os que têm a mesma raça do animal previamente selecionado. Mais adiante, vamos exibir como vai ser esse leiaute, mas, neste momento, vamos estudar a classe `ViewPetsCell`, que tem o código a seguir:

```
namespace App\View\Cell;

use Cake\View\Cell;
use Cake\ORM\TableRegistry;

class ViewPetsCell extends Cell
{
    protected $_validCellOptions = [];

    public function display( $pet )
    {
        $this->loadModel('Pets');

        if(isset($pet->breed)){

            $query = TableRegistry::getTableLocator()->get('Pets')-
>find('all', [
                'conditions' => [
                    'Pets.breed_id =' => $pet->breed->id,
                    'Pets.id <>' => $pet->id
                ],
                'contain' => ['Breeds'],
                'limit' => 10,
                'order' => ['views' => 'DESC']
            ]);
            $this->set('pets', $query->all());
        }else{
            $this->set('pets', []);
        }
    }
}
```

Uma classe `Cell` deve estender de `Cake\View\Cell`, cujo método principal é a *action* chamada `display`, que é onde devemos criar o corpo do código. Um detalhe é que também usamos a palavra *action* (assim como nos *controllers*) para os métodos que recebem e processam informações. Podemos até afirmar que uma classe `Cell` funciona como pequenos *controllers* que desempenham funções mais específicas. Como já mencionamos, embora seja uma classe para descrever alguma lógica, uma `Cell` também precisa de um template para apresentar as informações, e, seguindo a convenção que já conhecemos para os *controllers*, toda *action* tem um template correspondente. Logo, no exemplo descrito, precisamos ter um template para o método `display()`, então vamos criar um arquivo `display.php` em `templates/cell/viewPets/display.php`.

Uma *action* dentro de uma `Cell` pode acessar modelos e definir variáveis para o template usando o método `set()`. Na *action* `display()`, usamos `TableRegistry::getTableLocator()` para acessar o modelo `Pets` e fazer uma pesquisa com o método `find()`, que recebe como parâmetro a raça que deve ser retornada nos resultados, `'Pets.breed_id =' => $pet->breed->id`, além de limitar o retorno em 10 resultados o ordená-los de forma decrescente. No código do template `display.ctp`, vamos ter acesso à lista de pets retornados por `$this->set('pets', $query->all())`. Podemos, então, imprimir essas informações na página com o código a seguir:

```
<section>
    <header class="major">
        <span>Outros bichinhos da mesma raça</span>
    </header>
    <ul class="default">
        <?php foreach ($pets as $pet): ?>
            <li><?= $this->Html->link( h($pet->name), ['action' => 'view',
$pet->id]) ?> - <?= h($pet->age) ?> anos</li>
        <?php endforeach; ?>
    </ul>
</section>
```

É um exemplo simples: a *action* `display`, que fez uma pesquisa na tabela `pets`, retorna uma lista que foi armazenada em `$pets`, onde é usado um laço `foreach` para percorrer todos os itens e extrair o nome do *pet* para formar uma lista de *links* com o elemento *HTML* `<li>` e o *helper* `$this->Html->link`.

Feito tudo isso, o que ainda precisamos aprender é como inserir esta `Cell` em algum template. Isso pode ser feito com o método `$this->cell()`, como no exemplo de uso a seguir:

```
<?= $this->cell('ViewPets', [$pet]) ?>
```

Um detalhe que pode ser observado nesse trecho é que, assim como os *elements*, podemos passar dados para uma `Cell`. Nesse exemplo, a informação que passamos está na variável `[$pet]`, que é usada pela classe `ViewPetsCell`. O arquivo em que vamos adicionar esse trecho de código apresentado ainda não foi criado — é o template `view.php`, que vamos estudar no próximo tópico.

Página inicial AdPET

Neste momento do livro, temos o conhecimento necessário para melhorar o projeto AdPET. O que vamos finalizar agora é a página inicial do projeto e criar, também, a página de detalhe do pet, que é acessada quando a pessoa clica em algum bichinho exibido.

O primeiro ponto que vamos precisar verificar é o nosso arquivo de rotas, o `config/routes.php`. No capítulo 4, deixamos preparado esse arquivo com duas rotas, uma para a página inicial e outra para acessar a página de detalhes do pet, que são `$routes->connect('/')` e `$routes->connect('/pet/:id')`, respectivamente. A seguir, temos o código necessário para essa configuração funcionar:

```
Router::scope('/', function (RouteBuilder $builder) {


    $builder->connect('/',
        ['controller' => 'Pets', 'action' => 'index']);
```

```
    $builder->connect('/pet/:id',
        ['controller' => 'Pets', 'action' => 'view'],
        [
          'pass' => ['id'],
           'id' => '[0-9]+'
        ]);

    $builder->fallbacks('DashedRoute');
});
```

As rotas configuradas acessam a classe `PetsController`, cujas *actions* `index` e `view` precisam dos templates `index.php` e `view.php`, respectivamente. O esqueleto do *controller* foi criado no capítulo 6, mas vou repeti-lo a seguir, para não quebrar a nossa linha de raciocínio e porque vamos precisar melhorá-lo. A classe `PetsController` está a seguir:

```
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;


class PetsController extends AppController
{
    public function index(){

    }

    public function view($id = null){

    }
}
```

Por enquanto, esse *controller* é bem simples: temos uma *action* para representar a página inicial e outra para exibir detalhe de informações sobre o pet. Vamos começar a implementar o código para a *action* `index()` , que, quando acessada, queremos que exiba os pets já cadastrados no sistema pelos usuários. Considere que já temos informações cadastradas em nossa base (falaremos em outros capítulos como cadastrar dados pelo AdPET), então vamos ao código:

```
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;


class PetsController extends AppController
{

   public $paginate = [
        'limit' => 9
    ];

    public function initialize(): void
    {
        $this->loadComponent('Paginator');
    }

    public function index(){
        $this->Flash->set('Quem ama as criaturas de Deus, ama o próprio
Deus', ['element' => 'intro']);
        $pets = $this->paginate();
        $this->set(compact('pets'));
    }

    public function view($id = null){
```

```
    }
}
```

Uma mudança importante no código da *action* `index` é que não usamos a classe `TableRegistry`, que estudamos no capítulo 8, para trazer do banco uma coleção de registros. Aqui, usamos o método `$this->paginate()`, acessível no *controller* quando carregamos o componente `Paginator` em `$this->loadComponent('Paginator')`. A intenção aqui é paginar os resultados vindos do banco de dados e não trazer toda a carga de dados — pelos motivos que já sabemos. Ainda nesse método, usamos o `FlashComponent` para enviar uma mensagem para o template `index.php`.

Uma observação importante antes de codificar o arquivo `index.php` é que precisamos ter certeza de que o arquivo `templates/layout/default.php`, que é o leiaute principal do projeto, esteja finalizado corretamente com os arquivos CSS importados e com a estrutura básica implementada. A construção desse arquivo já foi estudada no tópico sobre leiaute. **Os arquivos CSS para o AdPET podem ser baixados em https://github.com/jozecarlos/adpet.** Não é objetivo do livro abordar este assunto, então vamos apenas usar estes arquivos para definir o visual do projeto AdPET.

Como sabemos, a estrutura do leiaute é bem simples, formada por alguns `Element`, mas para o leiaute funcionar corretamente, ainda falta criarmos todos os elementos usados no `default.php`. O `element` `templates/element/topbar.php` vai conter a estrutura do topo da página que possui a logo e o menu principal da aplicação. Seu código-fonte é:

```
<div id="header" class="skel-panels-fixed">
    <div class="container">
        <div id="logo">
            <h1><a href="/"><?php echo $this->Html->image('logo.png',
['alt' => 'AdPET']); ?></a></h1>
            <span class="tag">3.7</span>
        </div>
```

```
        <nav id="nav">
            <ul>
                <li class="active"><?= $this->Html->link('Home', '/' ); ?>
</li>

               <li><?= $this->Html->link('Cadastre-se', '/registration' );
?></li>

                <li><?= $this->Html->link('Login', '/login' ); ?></li>
            </ul>
        </nav>
    </div>
</div>
```

Por enquanto, vamos exibir três opções no menu: `Home` , `Cadastre-se` e `Login` . À medida que o projeto avança, mais itens vão ser adicionados ao menu. O *element* `templates/element/banner.php` tem como função exibir uma imagem mais atrativa na página inicial para melhorar a aparência e vai exibir a mensagem `Quem ama as criaturas de Deus, ama o próprio Deus` , enviada pelo `FlashComponent` na *action* `index()` .

```
<div id="banner" class="container">
    <section>
       <?= $this->Flash->render() ?>
    </section>
</div>
```

O arquivo `templates/element/copyright.php` vai ser responsável por exibir informações no rodapé da página. Este arquivo terá o seguinte código:

```
<div id="copyright">
    <div class="container">
        <div class="copyright">
            <p>Design: <a href="http://templated.co">
            TEMPLATED</a>
            Images: <a href="http://unsplash.com">Unsplash</a> (<a
href="http://unsplash.com/cc0">CC0</a>)</p>
```

```
            <ul class="icons">
                <li><a href="#" class="fa fa-facebook">
<span>Facebook</span></a></li>
                <li><a href="#" class="fa fa-twitter"><span>Twitter</span>
</a></li>
                <li><a href="#" class="fa fa-google-plus"><span>Google+
</span></a></li>
            </ul>
        </div>
    </div>
</div>
```

Este arquivo também é um `element` que vamos usar para exibir *links* para as redes sociais e também para exibir quem é o autor do design do template utilizado no projeto. O design do projeto AdPET foi adaptado de `http://unsplash.com/cc0` .

Se tudo foi feito como descrito até agora, o único passo que falta é terminar o `index.php` para finalizar a página inicial. Vamos ao código:

```
<div class="row no-collapse-1">

    <?php foreach ($pets as $pet): ?>
        <section class="4u pet_block">
            <?= $this->Html->image("/img/uploads/".h($pet-
>profile_picture), [
                "alt" => h($pet->name),
                "class" => "image featured",
                'url' => ['action' => 'view',  $pet->id]
            ]); ?>
            <div class="views">Visualizações: <?= $pet->views ?></div>
            <div class="box">
                <h3><strong><?= $this->Html->link( h($pet->name),
                ['action' => 'view', $pet->id]) ?></strong></h3>
                <p><?= h($pet->description) ?></p>
        </section>
    <?php endforeach; ?>

</div>
```

```
<div class="row" align="center">
        <ul class="page-buttons">
            <?php
            echo $this->Paginator->prev('« Previous');
            echo $this->Paginator->numbers();
            echo $this->Paginator->next('Next »');
            ?>
        </ul>
</div>
```

Nesse bloco de código, usamos o `foreach` para repetir a estrutura criada pelo elemento `<section>` para cada item da variável `$pets`, que foi criada no controle pela linha de código `$this->set(compact('pets'))`. Dentro do laço de repetição, é criado um *link* para redirecionar para a *action* `view($id)`, o que é feito pelo *helper* `$this->Html->link()`. A imagem cadastrada do pet vai ser exibida na página e usamos `$this->Html->image()` para criar o elemento `<img/>`. Sabemos que os registros dessa página foram paginados com nove itens, definidos na configuração do `PaginatorComponent` no `PetsController`. Os controles da paginação na página foram criados pelo `PaginatorHelper`, que adiciona botões para voltar e avançar e exibe botões para as páginas intermediárias. O resultado da página inicial do AdPET é:

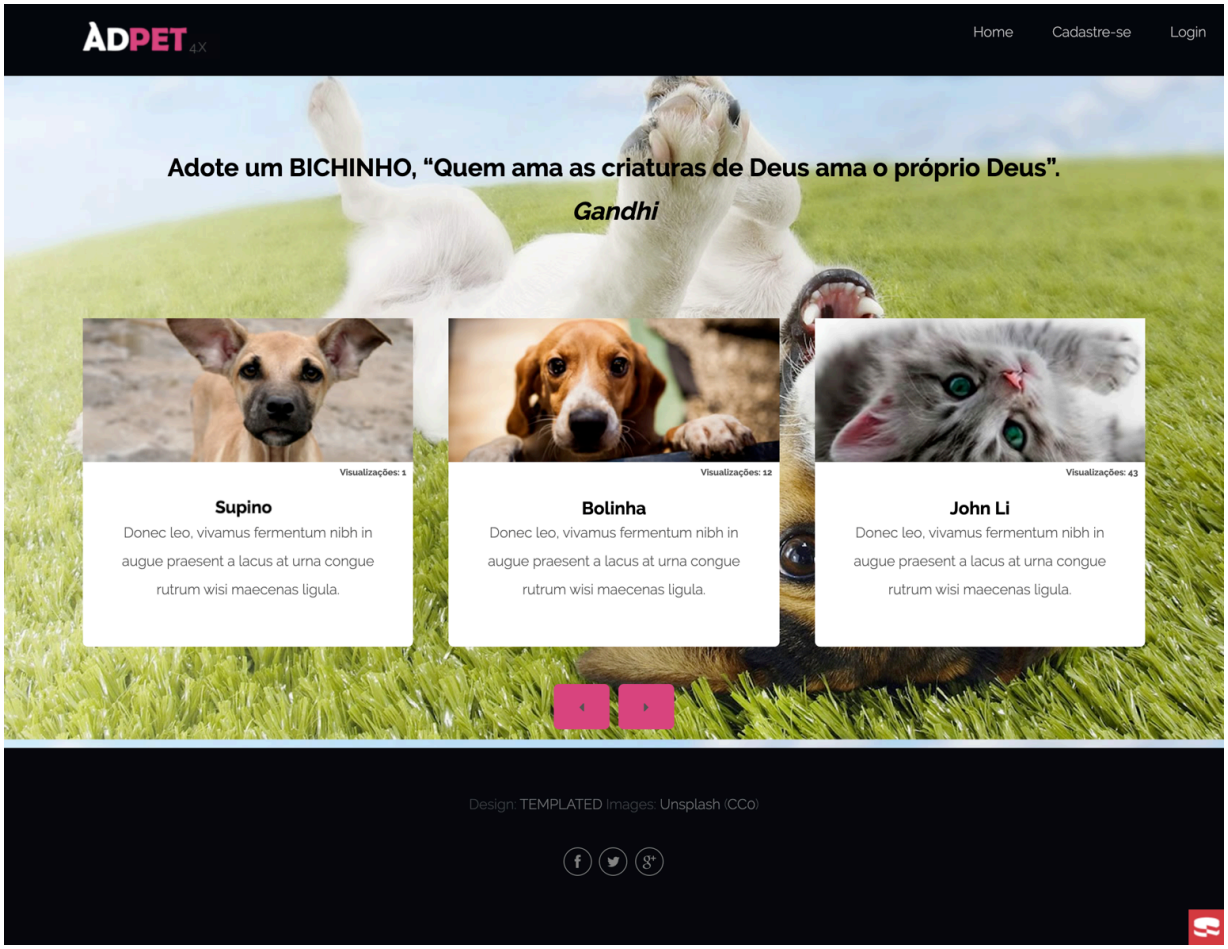


Figura 10.2: Página inicial do sistema AdPET.

Página de detalhe do pet

A página de detalhe do bichinho é acessada quando clicamos no nome de algum pet exibido na página inicial. Essa navegação é criada com esta linha de código: `$this->Html->link( h($pet->name), ['action' => 'view', $pet->id])`. Com ela, informamos para o *helper* que estamos criando um *link* que vai redirecionar para a *action* `view` e passamos como parâmetro o `id`. O texto do *link* vem do atributo `name`, da classe `Pet`. Vamos ver como será a página de detalhe de pet na imagem a seguir:

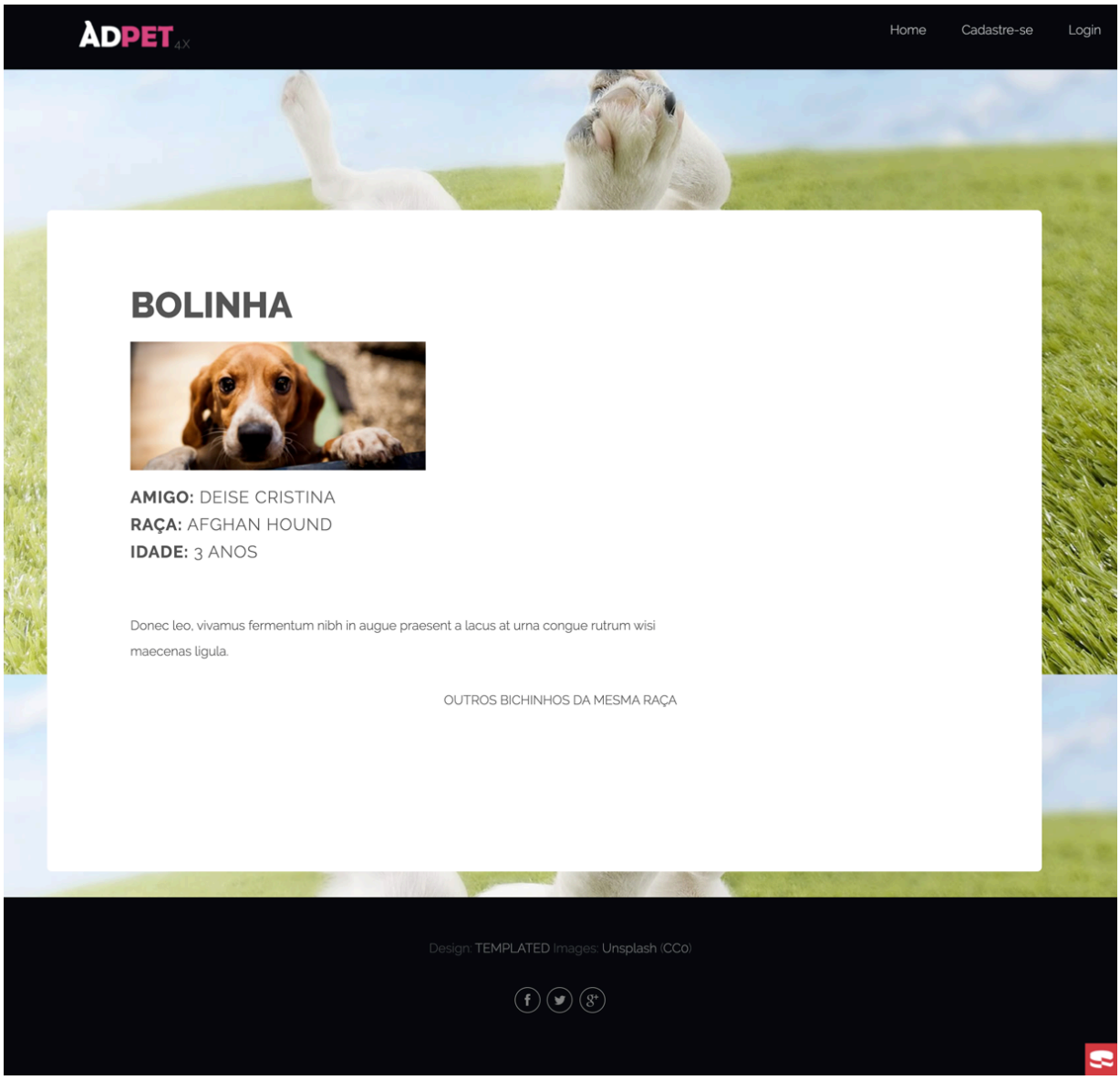


Figura 10.3: Página de detalhe do pet.

A imagem também exibe informações extras do pet, o que é tratado pela *action* `view` e o template `view.ctp`. No `PetsController`, precisamos editar `view($id = null)`. Esse método recebe uma requisição com o `id` do pet cujas informações pretendemos exibir. Uma das primeiras linhas que devemos escrever é como resgatar essas informações no banco de dados. Vamos direto ao código para entendermos melhor:

```php
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;
use Cake\ORM\TableRegistry;


class PetsController extends AppController
{

   public $paginate = [
        'limit' => 9
    ];

    public function initialize(): void
    {
        $this->loadComponent('Paginator');
    }

    public function index(){
        $this->Flash->set('Quem ama as criaturas de Deus, ama o próprio
Deus', ['element' => 'intro']);
        $pets = $this->paginate();
        $this->set(compact('pets'));
    }

    public function view($id = null){
      try{

            $pet = TableRegistry::getTableLocator()->get('Pets')->get($id,
              [
                  'contain' => ['Breeds','Users']
              ]);
            $this->set('pet', $pet);
        }
        catch(RecordNotFoundException $ex){
           $this->redirect(['action' => 'index'], 404);
        }
```

```
    }
}
```

Na *action* `view`, o parâmetro `id` é repassado durante a construção do link na página inicial, então usamos a classe `TableRegistry` para acessar o modelo `Pets` e chamar o método `get` para localizar o registro pelo `id`. Além de localizar as informações na tabela, queremos carregar junto as informações das tabelas relacionadas com `pets`, ou seja, carregar as informações dos modelos relacionados `Breeds` e `Users`. Nesse caso, queremos ter acesso às informações de raça e do usuário que cadastrou o pet, o dono do animal, que, na plataforma, chamamos carinhosamente de amigo.

Na *action*, temos um `try catch` para evitar `exceptions` na página para o usuário caso o `id` repassado não exista no banco de dados. Para finalizar, o registro retornado é enviado para a *view* por `$this->set('pet', $pet)`. Vamos analisar agora o arquivo `view.php`:

```
<div id="page" class="row">
    <div id="content" class="8u skel-cell-important">
        <section>
            <header class="major">
                <h2><?= h($pet->name) ?></h2>
                <br/>
                <div class="img">
                    <?= $this->Html->image("/img/uploads/" . h($pet-
>profile_picture), ['alt' => h($pet->name)]); ?>
                </div>
                <span class="byline">
                     <strong>Amigo:</strong> <?= h($pet->user->name) ?>
<br/>
                     <strong>Raça:</strong> <?= h($pet->breed->name) ?>
<br/>
                     <strong>Idade:</strong> <?= h($pet->age) ?> anos
</span>
            </header>
            <p><?= h($pet->description) ?></p>
        </section>
    </div>
```

```
    <div align="center" style="width: 100%">
        <?= $this->cell('ViewPets', [$pet]) ?>
    </div>
</div>
```

A estrutura HTML da página exibe os valores das propriedades do modelo `Pet`. Nesse caso, como se trata de um único registro, não é um objeto da classe `PetsTable`, e sim uma instância de `Entity`. Ainda nessa página, inserimos a `Cell` que estudamos e criamos neste capítulo. Ela recebe como parâmetro o registro `$pet` retornado pelo `PetsController` e, internamente, vai localizar mais registros relacionados com a mesma raça. Como dito, toda essa lógica da parte de buscar animais da mesma raça é tratada pela `Cell`, o que reforça o conceito de enquadrá-la como um componente que possui a sua própria lógica de negócio.

Conclusão

O CakePHP disponibiliza uma versatilidade admirável para facilitar o trabalho na camada de visão da aplicação. O uso de templates com diversas finalidades dá à pessoa desenvolvedora boas opções que facilitam o desenvolvimento de *views*. Neste capítulo, fizemos um apanhado geral de rotas, *controllers* e *helpers* para criar a página inicial do projeto AdPET. Nos próximos capítulos, vamos aprimorar ainda mais o projeto depois de aprendermos a salvar informação e, mais adiante, a autenticar os usuários do sistema.

Capítulo 11
Forms

Este é o capítulo em que vamos nos dedicar ao estudo sobre persistência de dados. O início dessa jornada já foi apresentado quando estudamos modelos. Um *model* tem como funcionalidade não apenas localizar informação, mas permitir que novos dados sejam criados, atualizados ou deletados. Ainda no decorrer do capítulo, vamos aprender a criar formulários com o uso de *helpers* e de uma classe especial chamada *modelless forms*.

11.1 Salvando dados

No CakePHP, o modo mais rápido de salvar informações é usando a classe `TableRegistry`, apresentada no capítulo 7, com a qual aprendemos mais contundentemente a pesquisar informações. O que ainda não aprendemos foi a usar essa classe para salvar informações. Para iniciar esse novo aprendizado, vamos para um exemplo prático, no qual criamos um registro novo no banco de dados. Imagine que queremos cadastrar um pet no sistema. Com a classe `TableRegistry`, teríamos:

```
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;
use Cake\ORM\TableRegistry;


class PetsController extends AppController
{
```

```
    public function save(){
        $petsTable = TableRegistry::getTableLocator()->get('Pets');
        $pet = $petsTable->newEntity();

        $pet->name = 'Layla';
        $pet->description = 'A cadela mais protetora do mundo';
        $petsTable->save($pet);
    }
}
```

Esse código não tem nada de complexo e acredito que, neste momento, já sabemos bem como é a filosofia do CakePHP. Primeiramente, criamos uma instância da classe `PetsTable` através de `TableRegistry::getTableLocator()->get('Pets')`, que cria um objeto da classe `Entity` usando o método `$petsTable->newEntity()`. Então atribuímos os valores para as propriedades pertinentes ao modelo e salvamos o registro no banco de dados com o método `save($pet)`. Obviamente, o exemplo tem apenas efeito didático — em termos práticos, no AdPET, que é um sistema web, as informações que serão inseridas no banco de dados virão de um formulário, em uma requisição `POST`. Nesse caso, pouca coisa muda, o que temos que fazer é iniciar um novo objeto `Entity` com as informações vindas via requisição `HTTP`. Vejamos como podemos fazer isso modificando o mesmo exemplo:

```
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;
use Cake\ORM\TableRegistry;


class PetsController extends AppController
{

    public function save(){
        $petsTable = TableRegistry::getTableLocator()->get('Pets');
```

```
        $pet = $petsTable->newEntity($this->request->getData());
         $petsTable->save($pet);
    }
}
```

Agora temos o objeto `$pet`, que foi criado com os dados vindo da requisição `HTTP` com o código `$pet = $petsTable->newEntity($this->request->getData())` — e não criado explicitamente com os dados diretamente no construtor. Essa é uma abordagem mais corriqueira quando estamos trabalhando com formulários web. Isso não significa que alguns atributos, dependendo da regra de negócio do sistema, não possam ser definidos diretamente via código, mas, para este momento, o que precisamos é saber que temos estas duas maneiras de criar um objeto e salvá-lo.

11.2 Atualizar e deletar dados

Ainda com a classe `TableRegistry`, podemos também modificar uma coleção de registros ou remover completamente o registro do banco de dados. Para atualizar ou deletar um registro, temos que trabalhar com um objeto criado a partir de informações salvas no sistema, e não criar um novo objeto. Por isso, o ponto de partida é localizar o registro no banco para, então, aplicar uma nova ação.

Por exemplo, a *action* `view($id = null)` em `PetsController`, criada no capítulo 10, recebe o parâmetro `id`, o qual é usado para localizar um registro e criar um novo objeto. Vamos melhorar esse código e avançar mais no nosso projeto de estudo. Vejamos, a seguir, o código já com algumas modificações:

```
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;
```

```
use Cake\ORM\TableRegistry;


class PetsController extends AppController
{

    public function view($id = null){
      try{
            $pet = TableRegistry::getTableLocator()->get('Pets')->get($id,
              [
                  'contain' => ['Breeds','Users']
              ]);
            $pet->views = $pet->views + 1;
            if ($this->Pets->save($pet)) {
                $this->set('pet', $pet);
            } else {
                $this->redirect(['action' => 'index'], 500);
            }
        }
        catch(RecordNotFoundException $ex){
           $this->redirect(['action' => 'index'], 404);
        }
    }
}
```

O que mudamos na *action* é que, toda vez que ela for acessada, vamos incrementar o valor da propriedade `views` da tabela `pets` pegando o valor anterior e somá-lo em 1. Essa informação vai ser útil para saber quantas vezes um pet foi visualizado. Podemos até usar essa informação para saber, entre os pets no sistema, quais estão despertando mais interesse dos usuários.

Continuando, depois de incrementar o valor, precisamos atualizar essa informação no banco, o que é feito novamente com o método `save()`, só que agora este método não está criando um registro, mas atualizando. O CakePHP sabe identificar se o objeto é uma instância nova ou um registro já existente no sistema.

Para deletar algum registro, seguimos a mesma lógica. É claro que somente vamos conseguir deletar um objeto que já foi persistido no sistema. Para exemplificar essa ação, criaremos uma nova *action* chamada `delete()`, que usaremos no capítulo final do livro. Vamos ao código:

```
<?php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Datasource\Exception\RecordNotFoundException;
use Cake\ORM\TableRegistry;


class PetsController extends AppController
{
    /// ... outras actions ocultadas...

    public function delete($id = null)
    {
        $petTable = TableRegistry::getTableLocator()->get('Pets');
        $pet = $petTable->get($id);

        if ($petTable->delete($pet)) {
            $this->Flash->success(__('O pet foi deletado'));
        } else {
            $this->Flash->error(__('O pet não pôde ser deletado'));
        }
        return $this->redirect("/index");
    }
}
```

No `delete()`, iniciamos com a localização do registro no banco de dados pelo `id`. O objeto criado com o retorno da pesquisa, `$pet`, vai ser deletado com o método `delete` em `$petTable->delete($pet)`. Essa operação pode retornar sucesso ou não; para cada estado, retornamos uma mensagem para o usuário com o `FlashComponent` e, caso a operação retorne sucesso, direcionamos para a página inicial com `return $this->redirect("/index")`.

Você deve ter observado que o ponto de partida para realizar qualquer das operações é usar a classe `TableRegistry`, seja para salvar um objeto novo, seja para atualizar ou deletar um registro do banco de dados.

11.3 Cadastro de usuários

No projeto AdPET, os usuários vão precisar entrar no sistema para divulgar um pet para adoção, ou para pedir para adotar. Para esses usuários realizarem alguma ação no sistema, primeiramente eles precisam se cadastrar; posteriormente, com as informações fornecidas no cadastro, eles precisam autenticar-se no AdPET. Por enquanto, vamos aprender como criar um formulário de cadastro no sistema e salvar as informações no banco de dados. No próximo capítulo, vamos aprender tudo sobre autenticação de usuários.

Quando abordamos o assunto sobre *migrations* no capítulo 3, criamos uma tabela chamada *users*. É essa tabela que vamos trabalhar no momento. Assumindo que a tabela foi devidamente criada no MySQL, a página inicial desenvolvida até agora tem no menu a opção `cadastre-se`. Esse link vai ser o nosso ponto de partida, onde o usuário, ao clicar, vai ser redirecionado para o formulário de cadastro.

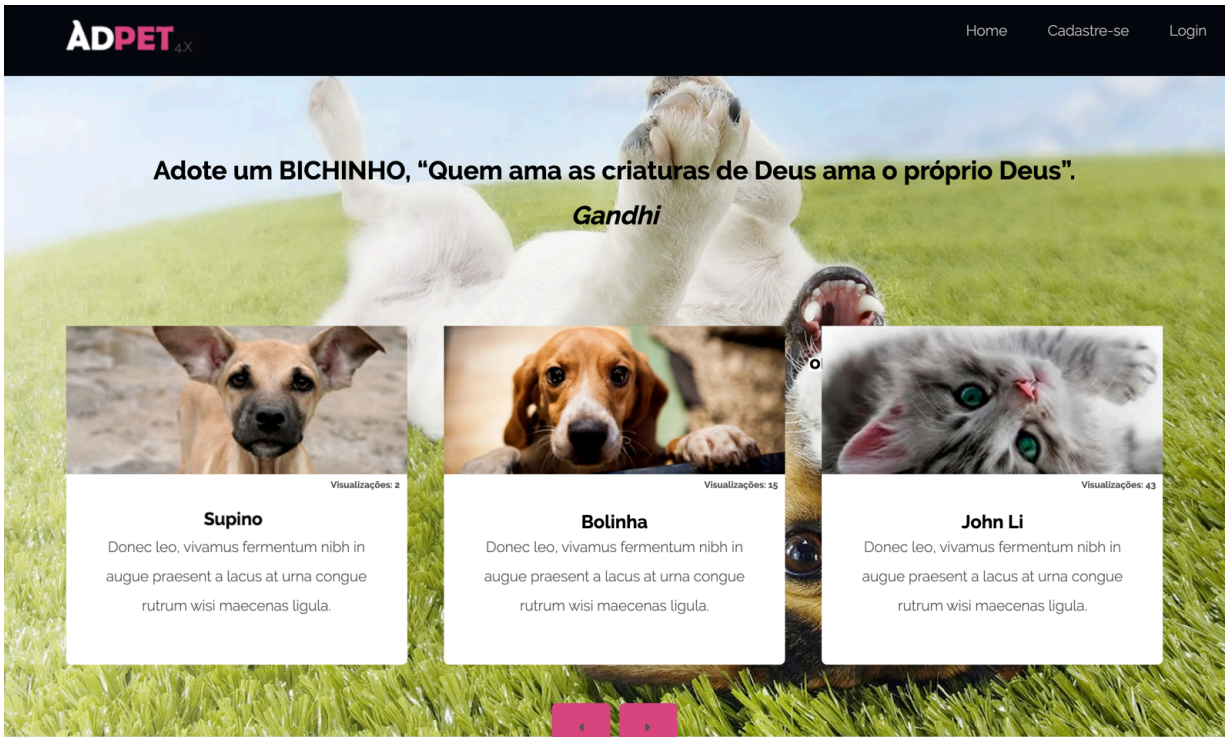


Figura 11.1: Página inicial do AdPET.

Temos nessa ação um redirecionamento: o usuário, ao clicar, é direcionado para a página de cadastro. Logo, o primeiro passo é verificar nossas rotas. Nesse caso, ainda não fizemos essa configuração que aponta para a página de cadastro, então precisamos criá-la. No arquivo `routes.php`, no diretório `config`, precisamos adicionar a seguinte instrução:

```
Router::scope('/', function (RouteBuilder $routes) {
    //...
    $routes->connect('/registration', [
        'controller' => 'Users',
        'action' => 'add']);
}
```

Nessa configuração estamos declarando que o caminho `/registration` vai ter sua requisição recebida no *controller* `UsersController` e na *action* `add`. Isso abre caminho para o nosso segundo passo, que é criar o *controller* `UsersController`. Até o momento, não tínhamos criado essa classe, que vai tratar de todas

as ações do usuário no sistema. Aqui é um bom ponto para relembrar como criar este arquivo usando o comando `bake`. Executaremos:

```
cake bake controller Users
```

Esse comando vai gerar a classe `UsersController` já com todos os métodos necessários para o CRUD — logo, a *action* `add` já foi escrita na classe e tem o seguinte corpo:

```
public function add()
    {
        $user = $this->Users->newEntity();
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request->getData());
            if ($this->Users->save($user)) {
                $this->Flash->success(__('O usuário foi salvo.'));

                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('O usuário não pôde ser salvo. Por favor, tente novamente.'));
        }
        $this->set(compact('user'));
    }
```

O código gerado acima já é totalmente funcional. Podemos até precisar fazer algumas mudanças nele, mas, agora, o que precisamos fazer é criar uma *view* para que esta *action* `add`, quando executada, não gere uma página de erro informando que o template não foi criado. Sempre seguindo a convenção do *framework*, o arquivo de que precisamos tem que ser criado no diretório `templates/User/add.php`. Com este arquivo, vamos adicionar o seguinte código:

```
<div class="basic_form">
    <h3>Registre-se para Adotar um Pet</h3>
```

```
    <?php
         echo $this->Form->create($user);
         echo    $this->Form->text('name',['placeholder' => 'Nome','label'
=> false]);
         echo    $this->Form->text('email',['placeholder' => 'E-mail',
'type' => 'email','label' => false]);
         echo    $this->Form->password('password',['placeholder' =>
'Senha', 'type' => 'password','label' => false]);
         echo    $this->Form->submit('Finalizar Cadastro',['class'
=>'button large']);
         echo $this->Form->end();
    ?>
</div>
```

Esse código cria o formulário para cadastrar usuários. Foi usado apenas o `FormHelper` para construir a página. Temos o `$this->Form->create($user)`, a primeira instrução que obrigatoriamente precisa ser usada. Esse método vai abrir uma tag `<form />`, que vai submeter os dados inseridos pelo usuário usando um *POST*. O código gerado por este método do `FormHelper` é o mesmo que escrever, usando HTML, a instrução `<form method="post" accept-charset="utf-8" action="/registration">`.

Em seguida, os *inputs* para inserir as informações são criados pelo método `$this->Form->text(string $name, array $options)`, onde determinamos o nome de cada campo e, no segundo parâmetro, definimos opções extras, como o *placeholder*. Finalizando, chamamos `$this->Form->end()` para fechar a *tag* `<form>` criada pelo `$this->Form->create($user)`. Só para comparar, se fôssemos escrever esse formulário apenas usando HTML, o código seria mais ou menos como o descrito a seguir:

```
 <div class="basic_form">
    <h3>Registre-se para Adotar um Pet</h3>
    <form method="post" accept-charset="utf-8" action="/registration">
        <input type="text" name="name" placeholder="Nome"
required="required">
        <input type="email" name="email" placeholder="E-mail"
```

```
required="required">
        <input type="password" name="password" placeholder="Senha"
required="required">
        <div class="submit">
        <input type="submit" class="button large" value="Finalizar
Cadastro">
        </div>
     </form>
</div>
```

Observando o exemplo acima, caso escolhêssemos criar um formulário sem o uso do *helper*, seria preciso escrever mais linhas e o código ficaria menos legível, embora o funcionamento do código permanecesse o mesmo. Para uma melhora na escrita de códigos, recomendamos o uso dos *helpers* no desenvolvimento das *views*, pois o código gerado será sempre menor e mais legível quando estamos trabalhando com CakePHP. Uma curiosidade: o `FormHelper` mudou a sua assinatura a partir da versão 3.4.0 do CakePHP. Por exemplo, antes, para criar um *input*, o método era `$this->Form->input(string $name, array $options)`.

Para finalizar, precisamos editar o menu da aplicação para chamar a página de cadastro. No capítulo 10, criamos um *element* chamado `topbar.php`. Esse arquivo é responsável por imprimir na página o menu do topo e a logo. Precisamos editá-lo para colocar o link para a rota `/registration` com a seguinte linha de código: `<?= $this->Html->link('Cadastre-se', '/registration' ); ?>`. Vale lembrar que, com este *helper*, `$this->Html->link(string $title, mixed $url = null, array $options = [])`, passamos o nome do link que ficará visível para o usuário e a URL que ele acessa — no caso, `Cadastre-se` e `/registration`, respectivamente. O resultado final para essa modificação será:

```
<div id="header" class="skel-panels-fixed">
   <div class="container">
       <div id="logo">
           <h1><a href="/"><?php echo $this->Html->image('logo.png',
```

```
['alt' => 'AdPET']); ?></a></h1>
            <span class="tag">3.7</span>
        </div>
        <nav id="nav">
            <ul>
                <li class="active"><?= $this->Html->link('Home', '/' ); ?>
</li>
                <li><?= $this->Html->link('Cadastre-se', '/registration'
); ?></li>
            </ul>
        </nav>
    </div>
</div>
```

Um lembrete para você: quando fizer o download das folhas de estilo do projeto, não se esqueça de colocá-las no diretório `webroot` do projeto AdPET. O link para download foi fornecido no capítulo anterior e é importante que você importe essas folhas de estilo caso não o tenha feito, para que a aplicação melhore visualmente com as mudanças que estamos fazendo no código do projeto. Ao atualizar a aplicação no browser e clicar no *link* `Cadastre-se` , vamos ter o seguinte resultado:

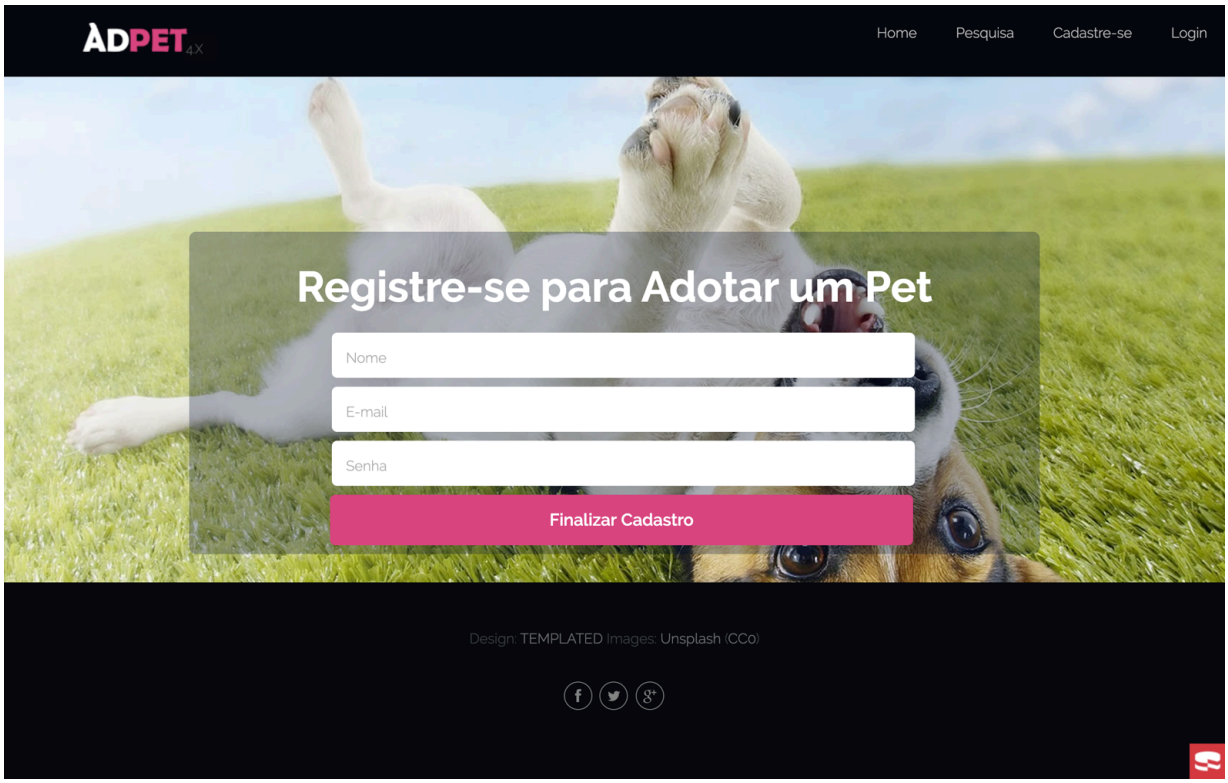


Figura 11.2: Página de cadastro dos usuários.

Até o momento, criamos tudo o que é necessário para cadastrar um usuário. Temos uma tabela no banco de dados para guardar as informações, criamos uma rota para acessar o *controller* e a *view* do formulário. Mas ainda podemos melhorar o método `add` utilizando *modelless forms*, que são *forms* cuja execução podemos controlar, aplicando validações e outros recursos bem antes de persistir a informação no banco de dados. Detalharemos mais no próximo tópico.

11.4 Modelless forms

Modelless forms são classes que estendem `Cake\Form\Form` e que permitem à pessoa desenvolvedora controlar a persistência de dados antes que sejam salvos no banco de dados. É um artefato útil para criar validações para as informações inseridas pelo usuário.

Por exemplo, para o cadastro do usuário, é fundamental adicionar validações para impedir que seja criado um usuário sem o e-mail ou uma senha com quantidade de caracteres insuficientes. Uma forma de validar informações dessa natureza é delegar a validação e a execução das informações para um *modelless form*. Vamos ao ponto. Criaremos um *form* chamado `UserRegisterForm` no diretório `src/Form` com o código a seguir:

```
 <?php
namespace App\Form;

use Cake\Form\Form;
use Cake\Form\Schema;
use Cake\Validation\Validator;
use Cake\ORM\TableRegistry;

/**
 * UserRegister Form.
 */
class UserRegisterForm extends Form
{
    /**
     * Builds the schema for the model less form
     *
     * @param Schema $schema From schema
     * @return Schema
     */
    protected function _buildSchema(Schema $schema)
    {
        return $schema->addField('name', 'string')
                      ->addField('email', ['type' => 'string'])
                      ->addField('password', ['type' => 'string']);
    }

    /**
     * Form validation builder
     *
     * @param Validator $validator to use against the form
     * @return Validator
```

```
     */
    protected function _buildValidator(Validator $validator)
    {
        return $validator
            ->add('name', 'length', [ 'rule' => ['minLength', 3],
'message' => 'A name is required'])
            ->add('email', 'format', ['rule' => 'email', 'message' => 'A
valid e-mail address is required'])
            ->add('password', 'length', [ 'rule' => ['minLength', 6],
'message' => 'The password is invalid, eg.: 123456']);
    }

    /**
     * Defines what to execute once the Form is being processed
     *
     * @param array $data
     * @return bool
     */
    protected function _execute(array $data)
    {
        $users = TableRegistry::getTableLocator()->get('Users');
        if ($users->save($users->newEntity($data))) {
           return true;
        }

        return false;
    }
}
```

Detalhando o código, uma classe *modelless form* precisa implementar 3 *hooks* para funcionar corretamente:

- `_buildSchema` : usado para criar o esquema de dados que será utilizado pelo `FormHelper` durante a criação do usuário;
- `_buildValidator` : método para implementar as validações dos atributos definidos no *schema*;
- `_execute` : método em que escreveremos a lógica para salvar os dados. Também é útil para reescrever algum comportamento antes de persistir dados.

No nosso exemplo, a classe `UserRegisterForm` define no método `_buildSchema` o esquema para o cadastro do usuário. Um esquema não é nada mais que um mapa de propriedades que precisam ser persistidas e cujos valores serão inseridos pelo usuário no formulário. Nesse caso, os atributos são `name`, `email` e `password`. No `_buildValidator`, definimos as regras que validam a informação submetida. O objeto `$validator` é uma instância da classe `Cake\Validation\Validator`, à qual adicionamos as seguintes regras:

- `name`: esse campo não pode ser vazio e deve ter, no mínimo, 3 caracteres;
- `email`: não pode ser nulo e o e-mail digitado tem que ser válido. A regra que determina se o e-mail é válido ou não já é implementada pelo validador;
- `password`: esse campo não pode ser vazio e precisa ter, no mínimo, 6 caracteres.

Definida a validação, no `_execute`, codificamos como as informações serão salvas. Para salvar as informações, é criado um objeto da classe `UserTable` com o método `TableRegistry::getTableLocator()->get('Users')` e elas são persistidas pelo método `$users->save($users->newEntity($data))`. Vale observar aqui que o objeto `users` é usado para persistir com o método `save` e também para criar um novo objeto em `$users->newEntity($data)`, que é uma instância da classe `Entity`, estudada no capítulo sobre modelos.

O *hook* `_execute` precisa retornar um `boolean` para indicar se os dados foram persistidos ou não. Agora que criamos e entendemos o uso do *modelless form*, vamos editar a *action* `add`, já criada anteriormente, e estudar as principais diferenças de implementação.

Modificando o add()

Quando criamos o `UsersController`, o fizemos com o uso do comando `bake`, que já adiciona todas as *actions* necessárias para o CRUD. A *action* `add` no *controller* está totalmente funcional e, caso o usuário queira testar o formulário de cadastro, vai funcionar perfeitamente.

Porém, não existe ainda validação alguma e, para resolver isso, vamos usar o *form* `UserRegisterForm` criado no tópico anterior. No entanto, antes de modificar, vamos estudar um pouco o código existente:

```
 public function add()
    {
        $user = $this->Users->newEntity();
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request-
>getData());
            if ($this->Users->save($user)) {
                $this->Flash->success(__('O usuário foi salvo.'));

                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('O usuário não pôde ser salvo. Por
favor, tente novamente.'));
        }
        $this->set(compact('user'));
    }
```

A primeira linha começa criando um novo objeto `$user` com a linha de código `$user = $this->Users->newEntity()` . Nesse exemplo, não declaramos explicitamente o uso da `UsersTable` , ou seja, quando acessamos `this->Users` , é o mesmo que usar `TableRegistry::getTableLocator()->get('Users')` , pois no *controller* temos também essa possibilidade.

Continuando, no código verificamos se a requisição feita vem a partir de um *POST*. Como esta operação é de um formulário de cadastro, os dados devem chegar via *POST*, que envia os valores no corpo da mensagem HTTP e não por URL, como é no caso do *GET*.

Quando a expressão `$this->request->is('post')` for verdade, a lógica continua atribuindo os valores que vêm na requisição para o objeto `$user` — lembrando que, para pegar os dados que vêm via *POST*, é

só usar o método `getData()` da classe *Request*, como em `$user = $this->Users->patchEntity($user, $this->request->getData())`. Em seguida, os dados são persistidos com `$this->Users->save($user)`. Caso essa operação retorne sucesso, o fluxo é direcionado para a *action* `index`. Se o retorno é falso, uma mensagem de erro é enviada para o usuário usando o *helper* `FlashHelper`. Esse é o método criado automaticamente quando usamos o `bake` — simples, mas bastante funcional.

Agora, vamos modificar a *action* e colocar nosso `UserRegisterForm` para aplicar algumas regras de validação.

```
  public function add()
    {
        $user = new UserRegisterForm();

        if ($this->request->is('post')) {
            if ($user->execute($this->request->getData())) {
                $this->Flash->success(__('O usuário foi salvo.'));

                return $this->redirect("/");
            } else {
                $this->Flash->error(__('Não foi possível adicionar o
usuário.'));
            }
        }

        $this->set('user', $user);
    }
```

A primeira mudança observada no código da *action* `add` é que criamos um objeto `$user` da classe `UserRegisterForm`, e não uma instância da classe `Entity`, como podemos observar no código anterior da *action*. A segunda mudança é que, quando a informação vai ser persistida, não usamos `$this->Users->save($user)`, e sim o método `$user->execute($this->request->getData())`, implementado na classe `UserRegisterForm`. Vale ressaltar que o método `execute()`

somente vai persistir caso as informações repassadas sejam validadas (as validações foram descritas em `_buildValidator`). Então, se os dados forem salvos no banco de dados, o fluxo é direcionado para a página inicial.

O redirecionamento do fluxo da página nesse exemplo é uma mudança interessante para comentar. Se observarmos, antes, a linha `$this->redirect(['action' => 'index'])` mudava o fluxo para uma outra *action* do mesmo `UsersController`. Já no segundo caso, em `$this->redirect("/")`, estamos redirecionando para o rota `/` declarada no arquivo `routes.php`, em `$routes->connect('/', ['controller' => 'Pets', 'action' => 'index']);`, que muda o fluxo para a *action* `index` do `PetsController` e não do `UsersController`. Para finalizar, se a persistência não for realizada, uma mensagem de erro é enviada com `$this->Flash->error(__('Unable to add the user.'))`. A seguir, temos um exemplo do comportamento da página quando existe algum erro na validação ou na persistência.

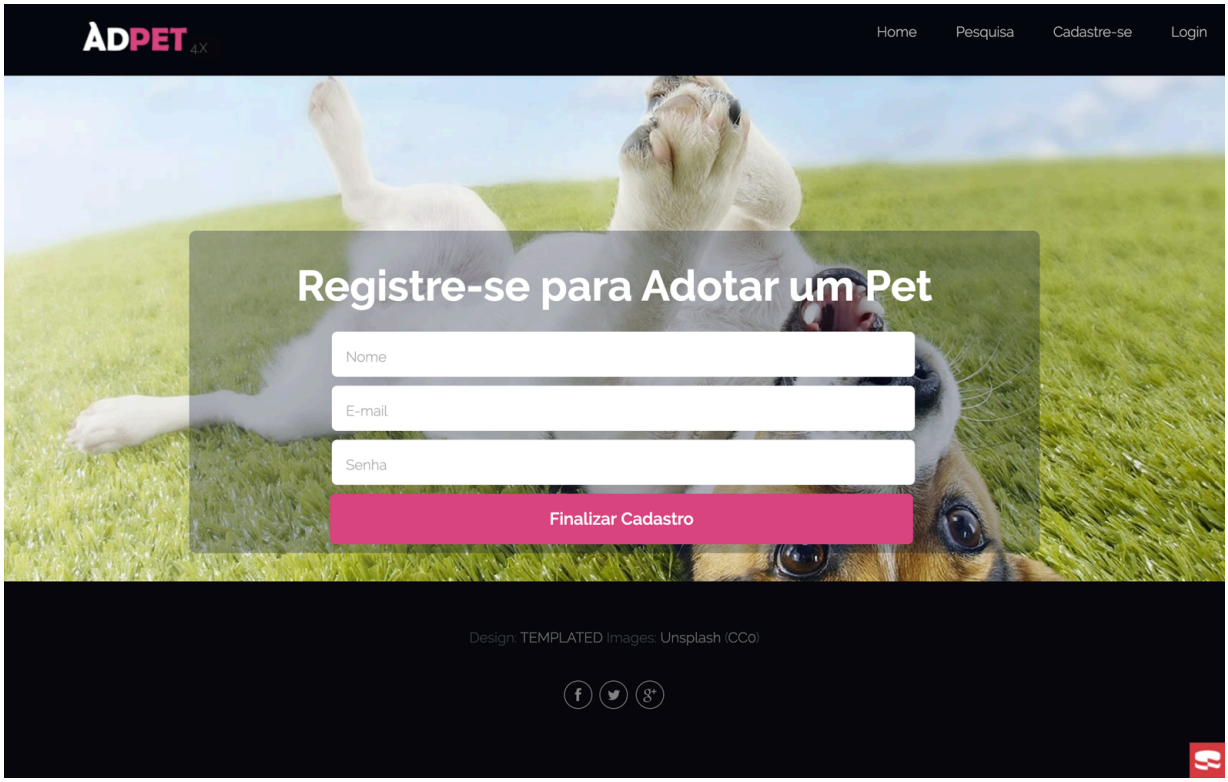


Figura 11.3: Página de cadastro dos usuários com erro.

Essa mensagem de erro na imagem é produzida pelo *helper* `FlashHelper`, apresentado no capítulo 9. Uma observação importante durante a construção do formulário `add.php` é que os nomes dos campos do formulário devem ser iguais aos nomes das colunas. Isso não é uma regra, mas é uma boa prática para facilitar a associação dos dados enviados pelo usuário com o modelo de dados. Com essa maneira de escrever, poupamos algumas linhas de código. Por exemplo, na tabela `users`, temos uma coluna chamada `name`, logo, criamos na *view*:

```
$this->Form->text('name',['placeholder' => 'Nome','label' => false]);
```

E no *schema* :

```
return $validator
            ->add('name', 'length', [ 'rule' => ['minLength', 3],
'message' => 'A name is required'])
```

Esse tipo de cuidado ao codificar vai facilitar associar os valores às colunas, como na linha `$this->Users->patchEntity($user, $this->request->getData())`, onde apenas atribuímos os valores para a `Entity`, passando diretamente os valores vindos de `$this->request->getData()`. Caso contrário, teríamos que tratar as informações antes de repassar. Por exemplo, se, em vez de `name` no formulário, usássemos `name_tmp` no *controller*, teríamos que manipular a informação, como no exemplo:

```
 public function add()
    {
        $user = $this->Users->newEntity();

        if ($this->request->is('post')) {
            $data = $this->request->getData();
            $user->set('name', $data['name_tmp'];
            $user->set('email', $data['email'];
            $user->set('password', $data['password'];

            if ($this->Users->save($user)) {
                $this->Flash->success(__('O usuário foi salvo.'));
```

```
                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('O usuário não pôde ser salvo. Por
favor, tente novamente.'));
        }
        $this->set(compact('user'));
    }
```

Ou seja, uma única linha, que era `$this->Users->patchEntity()`, virou mais quatro linhas de código só porque a propriedade `name_tmp` do *request* não corresponde a nenhuma propriedade do modelo `User`.

Conclusão

Neste capítulo, aprendemos como criar um formulário e salvar as informações no banco de dados, criando desde o acesso do formulário pela configuração de uma rota até a persistência, usando maneiras mais sofisticadas de validar a informação com *modelless form*.

No próximo capítulo, vamos trabalhar com a autenticação. Vamos criar um novo formulário, dessa vez não para persistir dados, mas para validar as credenciais do usuário inseridas durante o cadastro.

Capítulo 12
Autenticação

Segurança é uma parte fundamental de um sistema de informação. Por mais simples que seja a aplicação, geralmente é preciso identificar os usuários que pretendem acessar o sistema e proteger os dados de possíveis ataques e danos.

Normalmente, a forma mais usual de autenticação é usando um formulário que enviará credenciais, as quais serão validadas confrontando com dados previamente cadastrados no banco de dados. Com o CakePHP, seguindo poucos passos, é possível configurar um processo de autenticação eficiente para diferentes tipos de aplicações, formulários, APIs RESTful ou web services.

Antes de começar a exibir código, precisamos ter em mente quais passos precisam ser executados para um projeto CakePHP conseguir realizar autenticação. Se estamos falando em adicionar um processo em que vamos validar as credenciais de usuários do sistema, precisamos criar uma maneira de gerenciar esses usuários, guardando suas informações para posteriormente avaliá-las. Portanto, nosso primeiro passo vai ser configurar os serviços necessários para habilitar a autenticação no sistema.

12.1 Configurando o middleware

A partir da versão 3.X do CakePHP, foram introduzidos os *middlewares* e alguns processos passaram a ser tratados por essa camada, que pode ser até visualizada como uma classe *wrap*. Um *middleware* é, basicamente, uma camada lógica que intercepta

informações de outras partes da aplicação. O serviço de autenticação do CakePHP, por exemplo, deve passar a ser gerenciado através do *middleware* e a configuração precisa ser feita na classe `src/Application.php`.

Podemos dividir a configuração da autenticação em três passos: o primeiro é observar se o plugin *Authentication* está instalado na aplicação. Então, nosso ponto de partida é verificar se esse plugin já foi adicionado via `composer` no arquivo `composer.json` do projeto; caso não, basta executar o comando:

```
php composer.phar require cakephp/authentication:^2.0
```

Na classe `Application.php`, precisamos adicionar o plugin *Authentication* no método `bootstrap` usando o método `addPlugin`:

```
public function bootstrap()
{
    parent::bootstrap();
    ...
    $this->addPlugin('Authentication');
}
```

Muito cuidado para não deletar os outros plugins que provavelmente já estão configurados nessa classe — por exemplo, o plugin para as *migrations*. O código desse método vai ficar parecido com este:

```
public function bootstrap()
    {
        // Call parent to load bootstrap from files.
        parent::bootstrap();

        if (PHP_SAPI === 'cli') {
            try {
                $this->addPlugin('Bake');
            } catch (MissingPluginException $e) {
                // Do not halt if the plugin is missing
            }
```

```
            $this->addPlugin('Migrations');
        }

        /*
         * Only try to load DebugKit in development mode
         * Debug Kit should not be installed on a production system
         */
        if (Configure::read('debug')) {
            $this->addPlugin(\DebugKit\Plugin::class);
        }
        $this->addPlugin('Authentication');
}
```

Memorize esse passo, pois toda vez que instalarmos um novo plugin, vamos precisar carregá-lo no método `bootstrap` da classe `Application.php`. Tendo o plugin devidamente instalado e configurado, precisamos sobrescrever o método, e esta ação é o nosso segundo passo.

O método `getAuthenticationService` da classe `AuthenticationMiddleware`, uma classe interna do plugin `Authentication` que acabamos de carregar na aplicação, precisa ser reescrito com a lógica correta para funcionar com os atributos que o usuário vai fornecer para a autenticação das suas credenciais — que, no nosso projeto, são `email` e `password`:

```
  /**
     * @inheritDoc
     */
    public function getAuthenticationService(ServerRequestInterface
$request): AuthenticationServiceInterface
    {
        $service = new AuthenticationService();
        $service->setConfig([
            'unauthenticatedRedirect' => '/login',
            'queryParam' => 'redirect',
        ]);
```

```
        $fields = [
            'username' => 'email',
            'password' => 'password'
        ];

        // Load the authenticators, you want session first
        $service->loadAuthenticator('Authentication.Session');
        $service->loadAuthenticator('Authentication.Form', [
            'fields' => $fields,
            'loginUrl' => '/login'
        ]);

        // Load identifiers
        $service->loadIdentifier('Authentication.Password',
compact('fields'));

        return $service;
    }
```

O terceiro passo da nossa configuração é adicionar o `AuthenticationMiddleware` como um novo *middleware* da aplicação AdPET. A classe `Application` contém um método chamado `middleware`, onde informamos quais os *middlewares* que precisam ser usados na aplicação. Provavelmente esse método, em seu código local, já possui alguma implementação feita quando geramos a aplicação no início dos nossos estudos; logo, o que vamos fazer nesse caso é apenas adicionar mais algumas linhas de código:

```
   /**
     * Setup the middleware queue your application will use.
     *
     * @param \Cake\Http\MiddlewareQueue $middlewareQueue The middleware
queue to setup.
     * @return \Cake\Http\MiddlewareQueue The updated middleware queue.
     */
    public function middleware(MiddlewareQueue $middlewareQueue):
MiddlewareQueue
    {
        $authentication = new AuthenticationMiddleware($this);
```

```
        $middlewareQueue
          ->add(new ErrorHandlerMiddleware(Configure::read('Error')))
            ->add(new AssetMiddleware([
                'cacheTime' => Configure::read('Asset.cacheTime'),
            ]))
            ->add(new RoutingMiddleware($this))
            ->add($authentication);

        return $middlewareQueue;
    }
```

Nessa etapa, criamos um objeto da classe `AuthenticationMiddleware` passando como configuração o *controller* e a *action* que vai tratar as requisições de autenticação; em seguida, adicionamos esse objeto na *queue* `$middlewareQueue` .

Com o *setup* do serviço de autenticação pronto, precisamos verificar se a tabela que vai guardar as informações dos usuários está criada — a tabela `users` . Caso ainda não tenha essa tabela no seu banco de dados, reveja o capítulo 3, onde abordamos o estudo sobre as *migrations*. Lá foi criado o arquivo que vai gerar toda a estrutura da tabela. A seguir, vejamos o código da *migration* `CreateUsers` :

```
<?php
    use Migrations\AbstractMigration;

    class CreateUsers extends AbstractMigration
    {
      public function change()
      {
            $table = $this->table('users');

            $table->addColumn('name', 'string', [
              'default' => null,
              'limit' => 255,
              'null' => false,
            ]);
```

```
            $table->addColumn('email', 'string', [
              'null' => false,
            ]);

            $table->addColumn('senha', 'string', [
              'null' => false,
            ]);

            $table->addColumn('role', 'string', [
              'null' => false,
            ]);

            $table->addColumn('created', 'datetime', [
              'default' => null,
              'null' => true,
            ]);

            $table->create();
        }
    }
```

É uma tabela simples que vai guardar o nome do usuário, e-mail, senha e *role* — este último é o campo que identifica que tipo de usuário está acessando o sistema.

12.2 Autenticando nos controllers

No *controller*, é preciso carregar um componente chamado `authenticate`, responsável por disponibilizar como será o acesso às informações do usuário logado, definir rotas para o login e logout, entre outras funções. Esse componente deve ser inicializado na classe `AppController` no método `initialize`:

```
public function initialize()
{
    parent::initialize();

    $this->loadComponent('Authentication.Authentication', [
            'logoutRedirect' => '/users/login' // Default is false
    ]);
}
```

Com o método `initialize` configurado, precisamos criar duas novas *actions* com os nomes `login` e `logout` na classe `UsersController` — `login` para acessar ou autenticar as credenciais e `logout` para sair do sistema. Esta é a mesma classe criada quando estudamos como adicionar novos usuários no sistema.

```
<?php

namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\Event;
use Cake\Event\EventInterface ;

class UsersController extends AppController
{
    // Outros métodos do CRUD: add()..index()..

     public function beforeFilter(EventInterface $event)
    {
        parent::beforeFilter($event);

        $this->Authentication->allowUnauthenticated(['add', 'login',
'logout']);
    }

     public function login()
    {
        $result = $this->Authentication->getResult();
        if ($result->isValid()) {
```

```
            $target = $this->Authentication->getLoginRedirect() ??
'/index';
            return $this->redirect($target);
        }
        if ($this->request->is('post') && !$result->isValid()) {
            $this->Flash->error('E-mail ou senha inválida');
        }
    }

     public function logout()
    {
        $this->Authentication->logout();

        return $this->redirect(['controller' => 'Users', 'action' =>
'login']);
     }
}
```

No *controller* `UsersController` desse código, usamos o *listener* `beforeFilter(EventInterface $event)` para controlar se uma *action* deve ser acessada por uma requisição autenticada por um usuário ou não. A linha `$this->Authentication->allowUnauthenticated(['add', 'login', 'logout']);` está dizendo que todas as *actions* só vão ser executadas se a aplicação passou pelo processo de autenticação, enquanto as *actions* `login`, `logout` e `add` estão com acesso livre, ou seja, sem a necessidade de um processo de autenticação para acessá-las.

Na *action* `login`, o processo de autenticação das credenciais é feito pela linha de código `$this->Authentication->getResult()`, que recebe os dados vindos no POST. Não é necessário nessa parte o programador declarar as variáveis vindas na requisição. Se as credenciais forem válidas, a verificação `$result->isValid()` retornará verdadeiro e o método redirecionará para o último ponto do sistema antes de o usuário entrar na página de autenticação, ou para a página inicial do AdPET, caso seja o primeiro acesso do usuário no sistema. Se houver algum problema no processo de autenticação do usuário, uma mensagem de erro na página de login é exibida e nenhum redirecionamento ocorre.

Em `logout()`, a sessão autenticada do usuário é cancelada pelo método `$this->Authentication->logout()` e, em seguida, o fluxo é redirecionado para a página de login da aplicação, em `$this->redirect(['controller' => 'Users', 'action' => 'login'])`.

12.3 Página de login

O que foi feito até agora fica na camada de negócio. Configuramos o banco de dados, *controller*, *middleware*, porém ainda não existe nenhuma página para o usuário entrar com as suas credenciais — e é exatamente isso que vamos abordar agora.

O primeiro ponto que precisamos modificar é o arquivo `routes.php`. Devemos adicionar duas rotas: uma que redireciona para a *action* `login` e outra, para `logout`. O código que devemos adicionar é:

```
Router::scope('/', function (RouteBuilder $routes) {
    //...
    $routes->connect('/login', ['controller' => 'Users', 'action' =>
'login']);
    $routes->connect('/logout', ['controller' => 'Users', 'action' =>
'logout']);
}
```

Com essas duas linhas a mais no arquivo `routes.php`, deixamos acessíveis a página de login e uma forma de o usuário sair do sistema. Para a *action* `logout`, não vai ser preciso criar um template, pois a sua implementação redireciona o fluxo do sistema para a página de login, ou seja, não é preciso exibir nenhuma informação para deslogar do sistema. Na página de login, para coletar as credenciais, precisamos de um formulário para o usuário digitar e submeter as informações para análise. Para a *action* `login`, lembrando que sempre estamos seguindo a convenção para criar os nomes dos templates, vamos precisar criar o arquivo em `templates/User/login.php` com o seguinte código:

```
<div class="basic_form">
    <h3>Já possui um login</h3>
    <?php
        echo $this->Form->create();
        echo    $this->Form->email('email',['placeholder' => 'E-mail']);
        echo    $this->Form->password('password',['placeholder' =>
'Senha']);
        echo    $this->Form->submit('Acessar',['class' =>'button large']);
        echo $this->Form->end();
    ?>
</div>
```

O formulário de login é criado usando o `FormHelper` e possui dois campos. Um é para o e-mail, feito na linha:

```
$this->Form->email('email',\['placeholder' => 'E-mail']);
```

O outro é para senha, em:

```
$this->Form->password('password',['placeholder' => 'Senha']);
```

Quando o formulário for submetido, os dados serão enviados para a mesma *action* associada ao template `login.php` por padrão. Em `login()`, toda a lógica vai acontecer como descrevemos no tópico anterior. A seguir, podemos conferir a aparência da página de login do projeto AdPET:

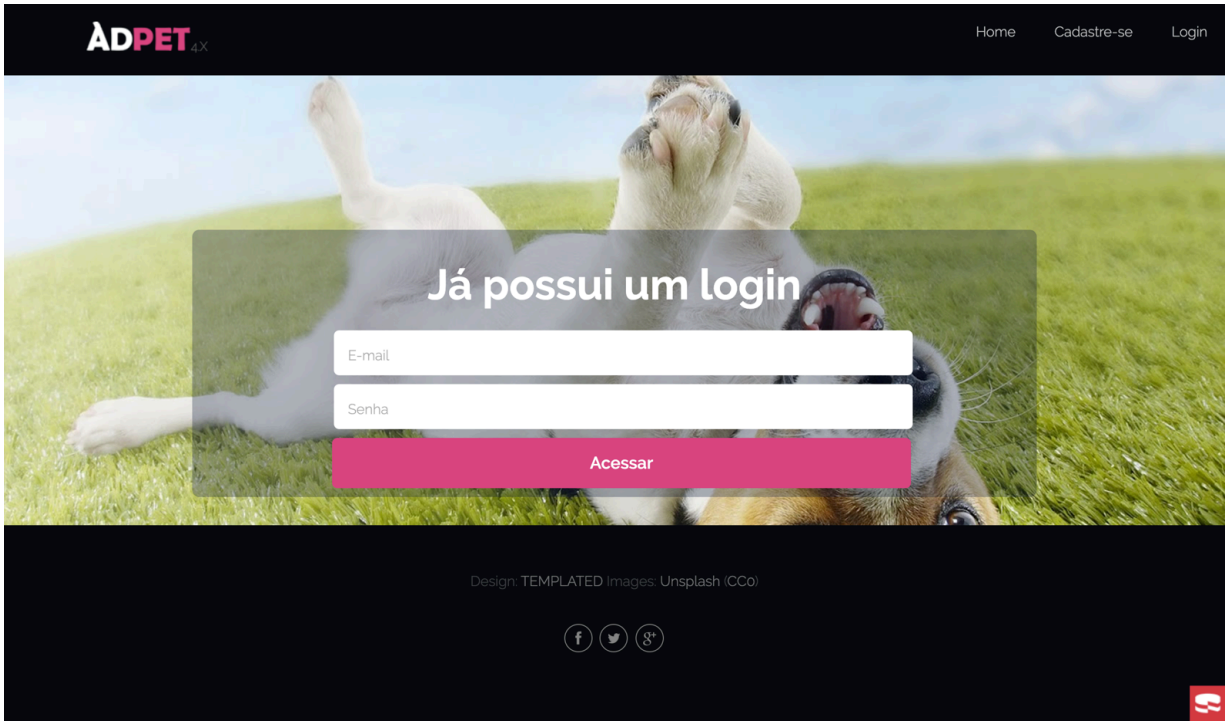


Figura 12.1: Página de login do AdPET.

12.4 O objeto Identity

É um objeto retornado pelo serviço de autenticação que provê acesso às informações do usuário logado no sistema. Quem retorna as informações é o método `getIdentifier()`, que pode ser acessado por quatro diferentes maneiras:

- Como um serviço: `$authenticationService->getIdentity()->getIdentifier()`;
- Por componente: `$this->Authentication->getIdentity()->getIdentifier()`;
- Em uma requisição: `$this->request->getAttribute('identity')->getIdentifier()`;
- Na *view*, como *helper*: `$this->Identity`.

O `$this->Identity` é um *helper* que pode ser acessado em um arquivo template, porém, é preciso inicializar o objeto na classe `AppView` localizada em `src/View/AppView.php` com a instrução `$this->loadHelper('Authentication.Identity')`. Veja o exemplo a seguir:

```
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{

    public function initialize()
    {
        $this->loadHelper('Authentication.Identity');
        $this->loadHelper('Paginator', ['templates' => 'paginator-
templates']);
    }
}
```

Já o objeto retornado pelo `getIdentifier()` consegue acessar as propriedades do usuário logado pelo método `get()` ou diretamente, por uma propriedade da classe `User`. Por exemplo, digamos que queremos acessar na *action* `view` o `id` do usuário logado. Nesse caso, podemos fazer:

```
<?php

namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\Event;

class UsersController extends AppController
{
    public function view()
    {
        $userId = $this->Authentication->getIdentity()->id;
```

```
    }
}
```

No template, o *helper* também acessa informações do usuário logado e dispõe de um método bem útil para verificar se existe uma sessão autenticada no sistema, o `isLoggedIn()`. Isso vai permitir controlar quais elementos podemos ou não exibir na página, levando em consideração se já temos ou não um usuário autenticado. Por exemplo, vamos modificar o arquivo `topbar.php` para restringir acesso a alguns botões do menu e exibir o nome do usuário logado no sistema. O código já modificado do `topbar.php` fica assim:

```
<div id="header" class="skel-panels-fixed">
    <div class="container">
        <div id="logo">
            <h1><a href="/"><?php echo $this->Html->image('logo.png',
['alt' => 'AdPET']); ?></a></h1>
            <span class="tag">3.8</span>
            <?php if($this->Identity->isLoggedIn()): ?>
                <br/>
                <span class="tag">Bem vindo, <?php echo $this->Identity-
>get('name');  ?></span>
            <?php endif; ?>
        </div>
        <nav id="nav">
            <ul>
                <li class="active"><?= $this->Html->link('Home', '/' ); ?>
</li>

                <?php if(!$this->Identity->isLoggedIn()): ?>
                    <li><?= $this->Html->link('Cadastre-se',
'/registration' ); ?></li>
                <?php endif; ?>

                <?php if(!$this->Identity->isLoggedIn()): ?>
                    <li><?= $this->Html->link('Login', '/login' ); ?></li>
                <?php endif; ?>

                <?php if($this->Identity->isLoggedIn()): ?>
```

```
                    <li><?= $this->Html->link('Sair', '/logout' ); ?></li>
                <?php endif; ?>
            </ul>
        </nav>
    </div>
</div>
```

Com esse código, os *links* `Cadastre-se` e `Login` vão aparecer agora como opções se não existir usuário logado no sistema. Os outros links do menu, `Home` e `Sair`, serão exibidos quando o usuário estiver autenticado no sistema. Esse comportamento foi inserido devido ao controle lógico usando `if` e o retorno da instrução `$this->Identity->isLoggedIn()`.

Ainda nesse template, caso o usuário esteja logado, vai ser impresso na página o nome lido pelo atributo `name` em `$this->Identity->get('name')`. Esse novo formato de código para o arquivo `topbar.php` será mantido e reutilizado no próximo capítulo usando a mesma lógica explicada neste tópico.

Conclusão

O presente capítulo apresentou toda a configuração necessária para permitir que o usuário se autentique no sistema. Esse é um passo importante para a segurança do projeto. Isso vai nos permitir criar áreas de acesso exclusivo para o usuário, o que será tema do próximo e último capítulo, no qual vamos finalizar o projeto AdPET.

Capítulo 13
Finalizando o Projeto AdPET

O AdPET é um projeto que tem a finalidade de criar uma ferramenta para a adoção de animais domésticos, carinhosamente chamados de pets. É uma escolha perfeita para introduzir o aprendizado do CakePHP de forma simples, divertida e prática. Posso também afirmar que existe um apelo social, se considerarmos que a nossa sociedade é deficiente em projetos e políticas públicas para cuidar de animais sem donos. Com este projeto, expresso um incentivo, pois, independente da profissão, sempre podemos fazer alguma coisa para ajudar a sociedade e os animais. Por exemplo, nós, como programadores e programadoras, podemos usar nossos conhecimentos técnicos para criar aplicações com esta finalidade, assim como o AdPET.

Durante os capítulos anteriores, sempre que foi introduzido um novo tópico sobre o framework CakePHP, usamos o nosso projeto de exemplo para aplicar e aprender da melhor maneira, dentro de um contexto sobre a funcionalidade em questão. Ao longo de 12 capítulos, chegamos até aqui com uma boa estrutura do projeto feita, desde a definição do banco de dados, autenticação e leiaute principal da aplicação.

Neste capítulo, nossa missão vai ser finalizar as funcionalidades do projeto, ao mesmo tempo que revisamos tudo o que aprendemos até aqui.

13.1 Meus pets

Como usuário do sistema AdPET, você pode enviar solicitações para adotar um pet ou cadastrar seus próprios pets para adoção. Essa

regra de negócio torna necessário criarmos uma área administrativa para o usuário gerenciar os pets. No projeto AdPET, vamos chamar essa área de `Meus Pets`. O objetivo deste tópico será criar essa página.

O primeiro passo é criar uma rota para deixar esta área acessível. Precisamos também alterar a classe `PetsController` e criar uma *action* chamada `myPets`. Vamos configurar essas informações no `router.php`:

```
Router::scope('/', function (RouteBuilder $routes) {
    //...
    $builder->connect('/my-pets',
        ['controller' => 'Pets', 'action' => 'myPets']);
}
```

Com esta rota, precisamos criar um novo item no menu do sistema em `topbar.php` e garantir o acesso para essa sessão:

```
<?php if($this->Identity->isLoggedIn()): ?>
   <li><?= $this->Html->link('Meus Pets', '/my-pets' ); ?></li>
<?php endif; ?>
```

Na implementação da *action* `myPets()`, queremos localizar todos os pets cadastrados pelo usuário; logo, vamos precisar das informações do usuário logado no sistema. Com elas, podemos desenvolver o código da *action* desta maneira:

```
<?php
namespace App\Controller;

use App\Controller\AppController;


class PetsController extends AppController
{
    public function myPets(){
```

```
        $petsTable = TableRegistry::getTableLocator()->get('Pets');

        $requestAdoptionsTable = TableRegistry::getTableLocator()-
>get('RequestAdoptions');

        $pets = $petsTable->find()
            ->where(['user_id =' => $this->Authentication->getIdentity()-
>id]);

        $requestAdoptions = $requestAdoptionsTable->find()
            ->where(['pet_id IN' => $pets->extract('id')->toArray()]);

        $this->set('pets', $pets->all());;
        $this->set('requestAdoptionsCount', $requestAdoptions->count());

    }
}
```

Na primeira linha de código do método, criamos uma instância da classe `PetsTable` para permitir fazer as buscas necessárias na base de dados. No método de busca, precisamos passar o `id` do usuário logado para filtrar todos os pets cadastrados por ele, resultado obtido em:

```
$petsTable->find() ->where(['user_id =' => $this->Authentication-
>getIdentity()->id])
```

Essa consulta é possível porque a tabela `pets` tem um relacionamento com a tabela `users` (`pets` *belongs to* `users`), que é uma associação criada pelo campo `user_id` na tabela `pets` . O `user_id` é o valor da coluna `id` do usuário logado e é obtido com o `AuthComponent` em `$this->Authentication->getIdentity()->id` .

Essa associação entre as tabelas foi discutida no capítulo 8, mas vamos relembrar um pouco. Para o arquivo `PetsTable.php` , temos a seguinte configuração para associar os modelos:

```
<?php
```

```
namespace App\Model\Table;

use Cake\ORM\Table;

class PetsTable extends Table
{
    public function initialize(array $config): void
    {
        $this->addAssociations([
            'belongsTo' => [
                'Users' => ['className' => 'App\Model\Table\UsersTable'],
                'Breeds' => ['className' => 'App\Model\Table\BreedsTable']
            ],
            'hasMany' => ['Messages']
        ]);

        $this->addBehavior('Timestamp');
    }
}
```

Em resumo, a classe `PetsTable` cria um vínculo entre o usuário e uma raça e pode, ainda, ter inúmeras mensagens. Ainda em `myPets()`, também queremos saber quais os pedidos que os pets do usuário tiveram. Para saber informações dessa natureza, temos que consultar a tabela `request_adoptions`, que tem por função guardar os pedidos de adoção entre dois usuários. Essa lógica é implementada na linha:

```
$requestAdoptions = $requestAdoptionsTable->find()
            ->where(['pet_id IN' => $pets->extract('id')->toArray()]);
```

Estamos criando uma `query` que localiza todos os registros de adoção para os pets do usuário. Usamos a cláusula do MySQL `IN` para filtrar todos os registros que têm o valor de `pet_id` igual a um valor dentro da lista de `ids`, criada com a instrução `$pets->extract('id')->toArray()`.

Com a *action* criada, precisamos criar a *view* para exibir as informações que foram disponibilizadas pela chamada das linhas

`$this->set('pets', $pets->all())` e `$this->set('requestAdoptionsCount', $requestAdoptions->count())`. O arquivo `templates/Pets/my_pets.php` vai conter a seguinte estrutura de código:

```
 <div class="row no-collapse-1">
   <?php foreach ($pets as $pet): ?>
        <section class="4u pet_block">
            <?= $this->Html->image("/img/uploads/".h($pet-
>profile_picture), [
                "alt" => h($pet->name),
                "class" => "image featured",
                'url' => ['action' => 'view',  $pet->id]
            ]); ?>
            <div class="views">
                <?php if ($requestAdoptionsCount > 0): ?>
                    <?= $this->Html->link('Pedidos:
'.$requestAdoptionsCount,
                        ['controller' => 'RequestAdoptions', 'action' =>
'index', $pet->id]) ?> -
                <?php endif; ?>
                Visualizações: <?= $pet->views ?>
            </div>
            <div class="box">
                <h3><strong><?= $this->Html->link( h($pet->name),
['action' => 'view', $pet->id]) ?></strong></h3>
                <p><?= h($pet->description) ?></p>
                 <div class="actions">
                    <?= $this->Html->link('<i class="fa fa-trash-o"></i>',
['controller'=>'pets','action'=>'delete',$pet->id],
                        ['confirm'=>'Tem certeza que quer deletar o
pet?','class' => 'button','escape' => false]); ?>
                    <?= $this->Html->link('<i class="fa fa-edit">
</i>','/pet/edit/'.$pet->id, ['class' => 'button','escape' => false]) ?>
                </div>
            </div>
        </section>
    <?php endforeach; ?>

</div>
```

A estrutura da página é similar ao leiaute da página inicial, onde vemos os pets dispostos em uma grade. A diferença é que estamos exibindo apenas os cadastrados pelo usuário e indicando se há alguma solicitação de adoção. Nesta tela, também adicionamos botões que permitem que o usuário edite ou delete as informações do pet cadastrado. As linhas a seguir adicionam esses botões para cada pet exibido:

```
<?php
<div class="actions">
     <?= $this->Html->link('<i class="fa fa-trash-o"></i>',
['controller'=>'pets','action'=>'delete',$pet->id],
                         ['confirm'=>'Tem certeza que quer deletar o
pet?','class' => 'button','escape' => false]); ?>
     <?= $this->Html->link('<i class="fa fa-edit"></i>','/pet/edit/'.$pet-
>id, ['class' => 'button','escape' => false]) ?>
</div>
```

Para finalizar, precisamos editar o menu principal da aplicação para criar um link que acesse a área `Meus Pets`. O arquivo que precisa ser editado é o *element* `toolbar.php`. Adicionamos a ele a seguinte entrada:

```
<?php if($this->Identity->isLoggedIn()): ?>
   <li><?= $this->Html->link('Meus Pets', '/my-pets' ); ?></li>
<?php endif; ?>
```

Ao colocar esse pequeno trecho no arquivo, será criada mais uma opção no menu que vai redirecionar para a página, porém esta opção somente estará visível se o usuário estiver autenticado. Este teste é feito na linha `$this->Identity->isLoggedIn()`, que retorna se o usuário está ou não autenticado no sistema. O resultado final da página pode ser visualizado a seguir:

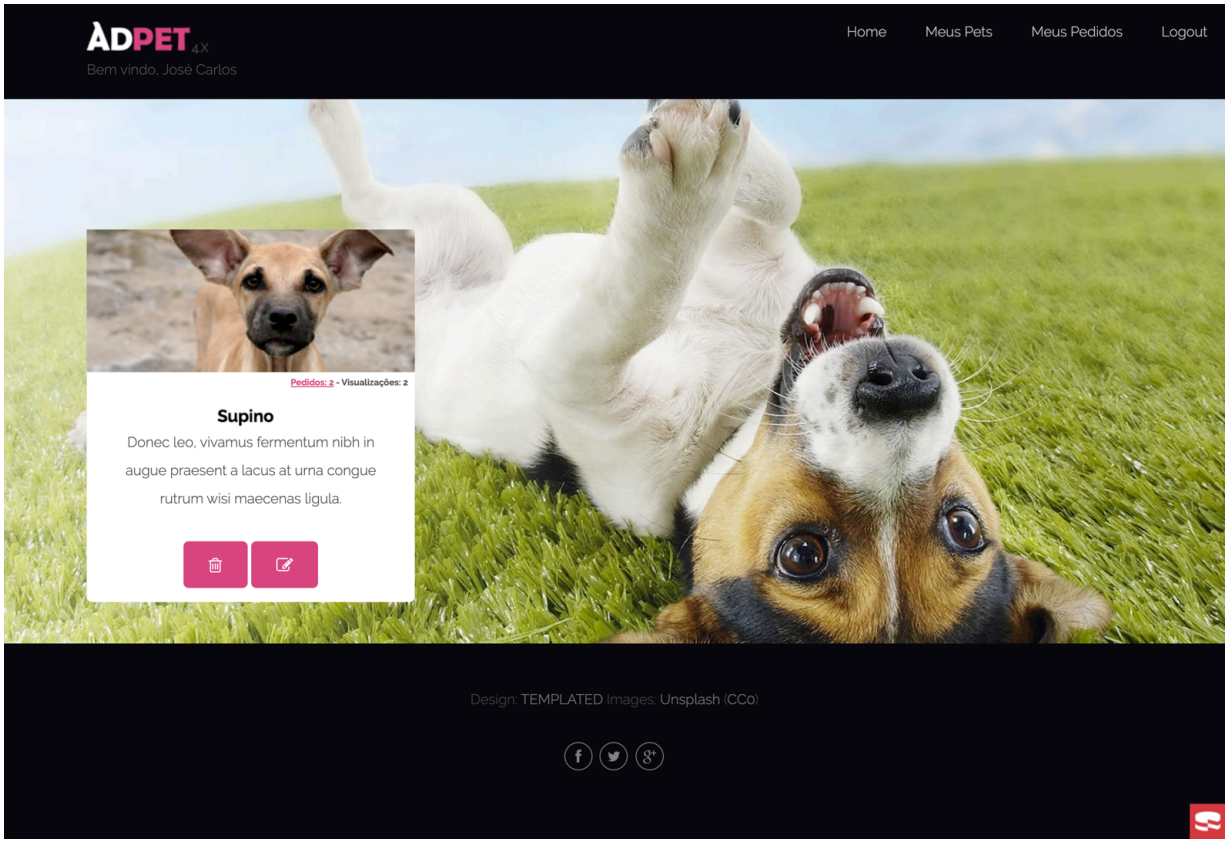


Figura 13.1: Página Meus Pets.

Cadastrando pets

A pré-condição para o cadastro de pets é que o usuário esteja autenticado na aplicação; sendo assim, se você ainda não configurou o projeto para que seja possível fazer a autenticação de usuários, volte ao capítulo 12, onde esse assunto é explicado. O acesso para a página de cadastro será pela área `Meus Pets`, arquivo `templates/Pets/my_pets.php`, que construímos no tópico anterior. Colocaremos uma seção adicional apenas para criar um link para a página de cadastro que ainda vamos construir:

```
<div class="row no-collapse-1">

    <section class="4u pet_block">
        <?= $this->Html->image("/img/dog_icon.png", [
            "alt" => 'Cadastro de Pet',
```

```
            "class" => "image featured",
            'url' => ['action' => 'add']
        ]); ?>
        <div class="views"></div>
        <div class="box">
            <h3><strong>Anunciar Pet</strong></h3>
            <p> </p>
            <div class="actions">
                <?= $this->Html->link('<i class="fa fa-plus">
</i>','/pet/add/', ['class' => 'button','escape' => false]) ?>
            </div>
        </div>
    </section>

   <?php foreach ($pets as $pet): ?>
        <section class="4u pet_block">
            <?= $this->Html->image("/img/uploads/".h($pet-
>profile_picture), [
                "alt" => h($pet->name),
                "class" => "image featured",
                'url' => ['action' => 'view',  $pet->id]
            ]); ?>
            <div class="views">
                <?= $this->Html->link('Pedidos: '.$requestAdoptionsCount,
                    ['controller' => 'RequestAdoptions', 'action' =>
'index', $pet->id]) ?>
                - Visualizações: <?= $pet->views ?>
            </div>
            <div class="box">
                <h3><strong><?= $this->Html->link( h($pet->name),
['action' => 'view', $pet->id]) ?></strong></h3>
                <p><?= h($pet->description) ?></p>
                 <div class="actions">
                    <?= $this->Html->link('<i class="fa fa-trash-o"></i>',
['controller'=>'pets','action'=>'delete',$pet->id],
                        ['confirm'=>'Tem certeza que quer deletar o
pet?','class' => 'button','escape' => false]); ?>
                    <?= $this->Html->link('<i class="fa fa-edit">
</i>','/pet/edit/'.$pet->id, ['class' => 'button','escape' => false]) ?>
```

```
                </div>
            </div>
        </section>
    <?php endforeach; ?>

</div>
```

Nesse arquivo, a seguinte linha de código adiciona um botão para o redirecionamento; porém, ainda não editamos o arquivo `routes.php` para deixar a navegabilidade funcionando.

```
<?= $this->Html->link('<i class="fa fa-plus"></i>','/pet/add/', ['class'
=> 'button','escape' => false]) ?>
```

Vamos adicionar mais uma rota para chamar a *action* `add` :

```
Router::scope('/', function (RouteBuilder $routes) {
    //...
    $builder->connect('/my-pets',
        ['controller' => 'Pets', 'action' => 'myPets']);
    $builder->connect('/pet/add', ['controller' => 'Pets', 'action' =>
'add']);
}
```

Nesse ponto da aplicação, a página já está funcionando. Claro que o botão de cadastrar vai dar um erro quando clicarmos nele, pois ainda não criamos o código que vai receber essa requisição. Se visualizarmos no navegador, veremos a seguinte tela quando clicamos em `Meus Pets` :

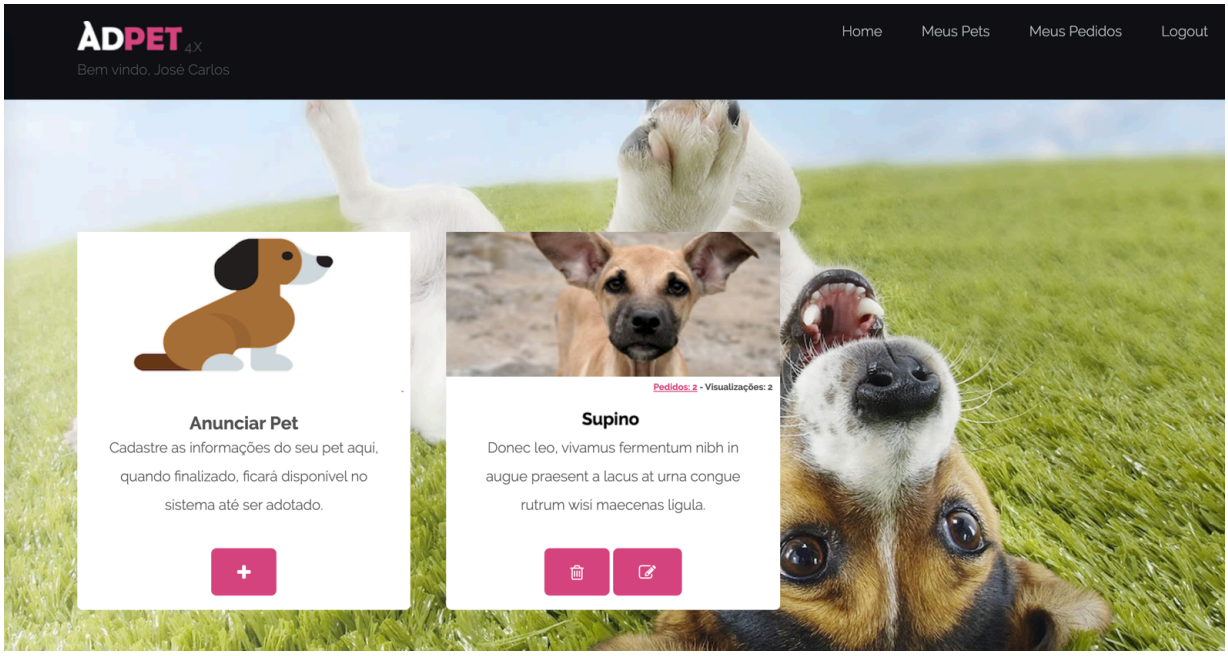


Figura 13.2: Página Meus Pets com o botão para adicionar um novo pet.

Em `PetsController`, a *action* `add` tem a seguinte lógica:

```
<?php
namespace App\Controller;

use App\Controller\AppController;


class PetsController extends AppController
{
    // manter as outras actions

    /**
     * Add method
     *
     * @return Response|null Redirects on successful add, renders view
otherwise.
     */
    public function add()
    {
        $petForm = new PetCreateForm();
```

```
        if ($this->request->is('post')) {

            $data = $this->request->getData();
            $attachment = $this->request-
>getUploadedFile('profile_picture');
            $data['profile_picture'] = $attachment->getClientFilename();

            if ($petForm->execute($data)) {
                $destination = WWW_ROOT.'img/uploads/'. $attachment-
>getClientFilename();
                $attachment->moveTo($destination);
                $this->Flash->success(__('O pet foi salvo.'));
                return $this->redirect("/my-pets");
            } else {
                $this->Flash->error(__('Não foi possível adicionar o
pet.'));
            }
        }

        $this->set('breeds', $this->Pets->Breeds->find('list'));
        $this->set('petForm', $petForm);
    }
  }
```

O código inicia criando um *modelless form* `PetCreateForm`. É esse `form` que vai processar as informações vindas do formulário e validá-las. Em seguida, testamos se a requisição vem de um `POST`; caso contrário, inicializamos duas variáveis que serão enviadas para `view`, `breeds` e `petForm`. Essa ideia de testar se é um `POST` ou não serve para aproveitar a mesma *action* que exibe a página de cadastro e processa as informações enviadas sem a necessidade criar *actions* separadas. Prosseguindo, quando a requisição entrar na condição `$this->request->is('post')`, os dados serão persistidos no banco de dados, em `$petForm->execute($data)`, se as informações repassadas em `$data = $this->request->getData()` estiverem corretas.

Ainda não apresentamos o código da classe `PetCreateForm`. Ela deve ser criada em `src/Forms/PetCreateForm.php` com o seguinte código:

```php
<?php
namespace App\Form;

use Cake\Form\Form;
use Cake\Form\Schema;
use Cake\Validation\Validator;
use Cake\ORM\TableRegistry;


/**
 * UserRegister Form.
 */
class PetCreateForm extends Form
{
    /**
     * Builds the schema for the model less form
     *
     * @param Schema $schema From schema
     * @return Schema
     */
    protected function _buildSchema(Schema $schema): Schema
    {
        return $schema->addField('name', 'string')
                      ->addField('description', ['type' => 'text'])
                      ->addField('gender', ['type' => 'string'])
                      ->addField('birthday', ['type' => 'datetime'])
                      ->addField('breed_id', ['type' => 'integer'])
                      ->addField('user_id', ['type' => 'integer']);
    }

    /**
     * Form validation builder
     *
     * @param Validator $validator to use against the form
     * @return Validator
     */
```

```
    protected function _buildValidator(Validator $validator): Validator
    {
        return $validator
            ->add('name', 'required', ['rule' => 'notBlank', 'message' =>
'The name can be empty'])
            ->add('gender', 'required', ['rule' => 'notBlank', 'message'
=> 'The gender can be empty'])
            ->add('birthday', 'required', ['rule' => 'notBlank', 'message'
=> 'The birthday can be empty']);
    }

    /**
     * Defines what to execute once the From is being processed
     *
     * @param array $data
     * @return bool
     */
    protected function _execute(array $data): bool
    {
        $pets = TableRegistry::getTableLocator()->get('Pets');
        if ($pets->save($pets->newEntity($data))) {
            return true;
        }

        return false;
    }
}
```

Em `_buildSchema` definimos as informações que devem ser repassadas pelo formulário e, em `_buildValidator` e `_execute`, temos a lógica para validar e salvar os dados respectivamente. Chegando a esse ponto, precisamos criar o arquivo `templates/Pets/add.php`, a `view` da *action* `add()` com o código:

```
<?php
<div class="basic_form">
<h3>Anunciar Pet</h3>
<?php
     echo $this->Form->create($petForm, ['type' => 'file']);
```

```
     echo    $this->Form->hidden('user_id', ['default' => $this->request-
>getAttribute('identity')->id]);
     echo    $this->Form->text('name',['placeholder' => 'Nome','label' =>
false]);
     echo    $this->Form->textarea('description', ['placeholder' => 'Fale
um pouco sobre o Pet','label' => false]);
     echo '<div class="date-picker">';
     echo    $this->Form->date('birthday', [
                 'templates' => [
                     'dateWidget' => '{{day}}{{month}}{{year}}',
                 ],
                'meridian'=> false,
                'hour'=> false,
                'minute'=> false,
                'second'=> false,
                'label' => [ 'class' => 'pet-form-label', 'text' => 'Data
de Nascimento']]);
     echo '</div>';
     echo    $this->Form->file('profile_picture', ['placeholder' => 'Foto
do Pet','label' => false]);
     echo    $this->Form->select('gender', ['M' => 'Macho', 'F' =>
'Fêmea'], ['placeholder' => 'Sexo','label' => false]);
     echo    $this->Form->select('breed_id', $breeds, ['placeholder' =>
'Raça','label' => false]);
     echo    $this->Form->submit('Finalizar Cadastro',['class' =>'button
large']);
     echo $this->Form->end();
?>
</div>
```

A página é apenas um formulário de cadastro dos pets. O código inicia usando o *helper* `FormHelper` com o objeto `$petForm`, que é uma instância da classe `PetCreateForm`. Em seguida, adicionamos todos os campos do formulário usando ainda métodos do *FormHelper*. Destaco a linha `$this->Form->date`, método que cria um campo que permite que o usuário selecione uma data a partir de um componente *datepicker*. Esse componente veio com a versão 4 do CakePHP.

Já em `$this->Form->file('profile_picture', ['placeholder' => 'Foto do Pet','label' => false])`, estamos criando um `input` para o usuário enviar um arquivo que vai ser a foto do perfil do animal. Para o upload funcionar, não podemos esquecer de adicionar a instrução `['type' => 'file']` em:

```
$this->Form->create($petForm, ['type' => 'file'])
```

Sem isso, o arquivo selecionado no formulário não vai funcionar. No *controller*, acessamos o arquivo usando o método:

```
$attachment = $this->request->getUploadedFile('profile_picture');
```

Com o objeto do arquivo representado pela variável `$attachment`, resgatamos o nome do arquivo para salvar no banco de dados e movemos para o diretório *uploads* em:

```
...
$data['profile_picture'] = $attachment->getClientFilename();
...
$destination = WWW_ROOT.'img/uploads/'. $attachment->getClientFilename();
               $attachment->moveTo($destination);
```

A variável `WWW_ROOT` é uma constante que representa o diretório `webroot` do CakePHP. É o mesmo diretório em que armazenamos os arquivos `CSS` e `JavaScript`. Com a *action* e o template do cadastro devidamente criados, a página de cadastro deve ter a seguinte apresentação:

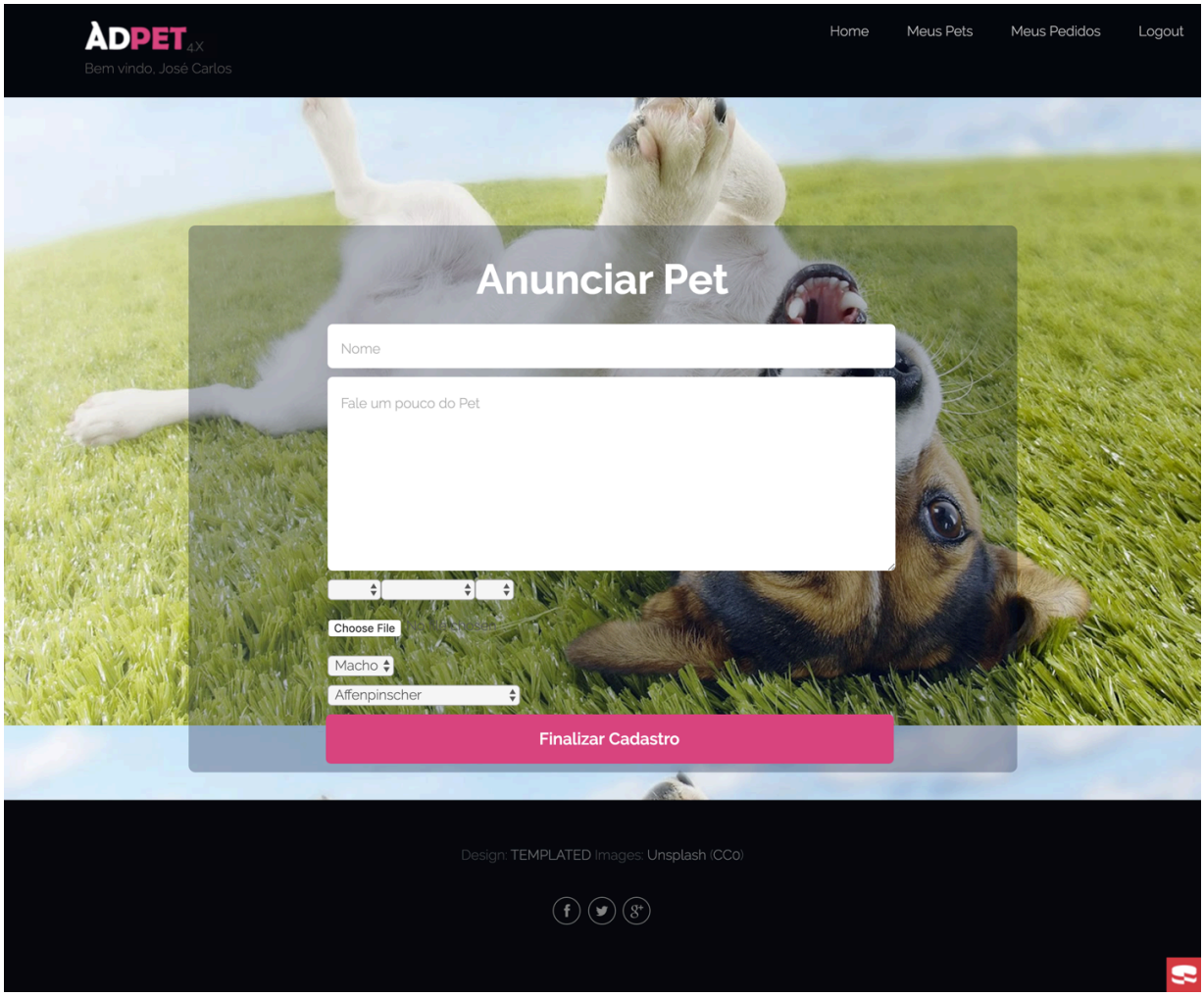


Figura 13.3: Formulário de cadastro de pets.

Editando e deletando pets

Editar ou deletar um pet são ações que também acontecem na seção `Meus Pets`. Esses controles já foram adicionados no projeto quando criamos o arquivo `my_pets.php`. O código que lista os pets cadastrados pelo usuário tem dois botões, um para deletar e outro para editar. Para o botão de deletar, utilizamos o seguinte código:

```
 <?= $this->Html->link('<i class="fa fa-trash-o"></i>',
['controller'=>'pets','action'=>'delete',$pet->id],
                        ['confirm'=>'Tem certeza de que quer deletar o
pet?','class' => 'button','escape' => false]); ?>
```

Esse botão de deletar tem uma peculiaridade. Quando clicado, ele mostra o seguinte diálogo de confirmação para o usuário: `Tem certeza de que quer deletar o pet?'` . Esse comportamento é criado quando utilizamos a propriedade `confirm` no `array` de configuração de atributos no *helper* `$this->Html->link()` . O outro link é para editar:

```
<?= $this->Html->link('<i class="fa fa-edit"></i>','/pet/edit/'.$pet->id,
['class' => 'button','escape' => false]) ?>
```

O que não fizemos para que as ações funcionem foi criar as devidas *actions* no controle, então vamos ao código. Para deletar um registro, criamos um método no capítulo 12. Vamos aproveitar o que aprendemos e refazer o método aqui, com pequenas alterações:

```
 /**
     * Delete method
     *
     * @param string|null $id Pet id.
     * @return \Cake\Http\Response
     * @throws \Cake\Datasource\Exception\RecordNotFoundException When
record not found.
     */
    public function delete($id = null)
    {
        $this->request->allowMethod(['get', 'delete']);

        $petTable = TableRegistry::getTableLocator()->get('Pets');
        $pet = $petTable->get($id);

        if ($petTable->delete($pet)) {
            $this->Flash->success(__('O pet foi excluído.'));
        } else {
            $this->Flash->error(__('Não foi possível excluir o pet. Por
favor, tente novamente.'));
        }
        return $this->redirect("/my-pets");
    }
```

Na primeira linha da *action*, definimos quais tipos de verbos `HTTP` podem prosseguir com a execução do método em `$this->request->allowMethod(['delete'])`. Nesse caso, estamos permitindo apenas o `delete`. As próximas linhas seguem igual ao que já fizemos — primeiro, encontramos o registro no banco de dados que precisamos deletar:

```
$petTable = TableRegistry::getTableLocator()->get('Pets');
        $pet = $petTable->get($id);
```

Em seguida, com o objeto `$pet` criado, passamos para a instância de `PetsTable` a exclusão do registro, na linha `$petTable->delete($pet)`. Caso a operação retorne sucesso, o fluxo da requisição é redirecionado para a mesma página. Com a ação de deletar concluída, para editar, vamos precisar criar mais uma *action* com o código a seguir:

```
/**
     * Edit method
     *
     * @param string|null $id Pet id.
     * @return \Cake\Http\Response|null Redirects on successful edit,
renders view otherwise.
     */
    public function edit($id = null)
    {
        $petForm = new PetCreateForm();
        $petTable = TableRegistry::getTableLocator()->get('Pets');
        $pet = $petTable->get($id);
        $profile_picture = $pet->profile_picture;

        if ($this->request->is('get')) {
            $petForm->setData([
                'name' => $pet->name,
                'description' => $pet->description,
                'gender' => $pet->gender,
                'birthday' => $pet->birthday,
                'breed_id' => $pet->breed_id,
            ]);
```

```
        }
        if ($this->request->is(['patch', 'post', 'put'])) {

            $data = $this->request->getData();
            $time =  Time::now()->year($data['birthday']['year'])
                ->month($data['birthday']['month'])
                ->day($data['birthday']['day']);
            $data['birthday'] = $time->i18nFormat('yyyy-MM-dd');

            if(strlen($data['profile_picture']) == 0){
                $data['profile_picture'] = $profile_picture;
            }

            TableRegistry::getTableLocator()->get('Pets')-
>patchEntity($pet, $data);

            if ($petTable->save($pet)) {
                $this->Flash->success(__('O pet foi salvo.'));
                return $this->redirect("/my-pets");
            } else {
                $this->Flash->error(__('O pet não pôde ser salvo. Por
favor, tente novamente.'));
            }
        }
        $this->set('breeds', $this->Pets->Breeds->find('list'));
        $this->set('petForm', $petForm);
    }
```

A base do código é bem parecida com `add()`. Nela, vamos usar ainda a classe `PetCreateForm()` e criar um outro arquivo template chamado `edit.php`, que vai ser idêntico ao arquivo `add.php` (é só criar o arquivo e copiar o conteúdo). Para editar um registro, o primeiro ponto é localizá-lo no banco de dados — esta tarefa é executada na linha `$petTable->get($id)`. Ainda no mesmo método, quando a página é requisitada, as informações do registro localizado são enviadas para a *view* depois que a instância de `PetCreateForm` é inicializada, com o valores resgatados, em:

```
if ($this->request->is('get')) {
            $petForm->setData([
                'name' => $pet->name,
                'description' => $pet->description,
                'gender' => $pet->gender,
                'birthday' => $pet->birthday,
                'breed_id' => $pet->breed_id,
            ]);
}
```

Quando os dados vêm do formulário, via `POST`, o fluxo segue o mesmo padrão já discutido em `add()` e os dados são lidos da requisição e atualizam o registro no banco de dados em `$petTable->save($pet)`. Para concluir esse fluxo, precisamos criar mais um registro no arquivo `route.php`:

```
$builder->connect('/pet/edit/:id', ['controller' => 'Pets', 'action' => 'edit'],['pass' => ['id'],'id' => '[0-9]+']);
```

Com `$pet = $petTable->get($id)`, estamos criando uma rota para editar o registro de pet no qual passamos como parâmetro o `id` cujo valor vai ser reutilizado pela *action* para localizar o registro na base de dados.

13.2 Solicitação de adoção

É sabido que o sistema AdPET permite que os usuários façam solicitações de adoção dos pets cadastrados no sistema. Essas solicitações vão partir da página inicial, que já havíamos trabalhado no capítulo 10. Neste tópico, vamos precisar modificar tanto o template como a *action* `index()` em `PetsController`. A proposta para essa tela é que qualquer usuário, autenticado ou não, veja uma lista de pets para adoção, porém somente os usuários autenticados conseguem ver um botão para solicitar uma adoção. Iniciaremos as alterações em `index.php`. Vejamos o novo formato do template:

```
 <div class="row no-collapse-1">

    <?php foreach ($pets as $pet): ?>
        <section class="4u pet_block">
            <?= $this->Html->image("/img/uploads/".h($pet-
>profile_picture), [
                "alt" => h($pet->name),
                "class" => "image featured",
                'url' => ['action' => 'view',  $pet->id]
            ]); ?>
            <div class="views">Visualizações: <?= $pet->views ?></div>
            <div class="box">
                <h3><strong><?= $this->Html->link( h($pet->name),
['action' => 'view', $pet->id]) ?></strong></h3>
                <p><?= h($pet->description) ?></p>
                <?php if ($this->Identity->isLoggedIn()): ?>
                     <?=
                        $this->Form->postButton('Eu quero', [
                                'controller' => 'RequestAdoptions',
                                'action' => 'add', $pet->id],  ['class' =>
'button']);
                     ?>
                <?php endif; ?>
            </div>
        </section>
    <?php endforeach; ?>

</div>
<div class="row" align="center">
        <ul class="page-buttons">
            <?php
            echo $this->Paginator->prev('« Previous');
            echo $this->Paginator->numbers();
            echo $this->Paginator->next('Next »');
            ?>
        </ul>
</div>
```

A principal mudança que adicionamos agora é o trecho a seguir:

```
<?php if ($this->Identity->isLoggedIn()): ?>
    <?= $this->Form->postButton('Eu quero', [
          'controller' => 'RequestAdoptions',
          'action' => 'add', $pet->id],  ['class' => 'button']);
    ?>
<?php endif; ?>
```

Nesse bloco de código, estamos restringindo o acesso ao botão para o usuário logado, checando com o `if ($this->Identity->isLoggedIn())`. O *helper* `$this->Form->postButton()` cria um botão junto a um `<form>` que vai submeter uma requisição `POST` direta para o *controller* `RequestAdoptions` sem a necessidade de usar o tradicional `$this->Form->create()`. A *action* que vai processar a informação é `add`, que terá a seguinte implementação:

```
use App\Form\RequestAdoptionMessageForm;
use Cake\Event\Event;
use Cake\ORM\TableRegistry;


class RequestAdoptionsController extends AppController {

    public function add($petId){

        $userId = $this->Authentication->getIdentity()->id;
        $adoption = $this->RequestAdoptions->newEmptyEntity();

        $countRequestAdoption = $this->RequestAdoptions
            ->find()->where([
                'pet_id' => $petId,
                'user_id' => $userId,
        ])->count();

        if($countRequestAdoption == 0){
            $adoption->user_id = $userId;
            $adoption->pet_id = $petId;

            if($this->RequestAdoptions->save($adoption)){
                $this->Flash->success(__('Solicitação de adoção
```

```
enviada.'));
            }
        } else {
            $this->Flash->error(__('Você tem uma solicitação de adoção
para esse pet.'));
        }

        return $this->redirect($this->referer());
    }
}
```

`RequestAdoptionsController` é o *controller* responsável por implementar tudo relacionado ao modelo de negócio para `RequestAdoptions`. Na *action* `add()`, a primeira instrução é para resgatar o `id` do usuário logado. Essa informação é importante para persistir o objeto `$adoption`. O trecho a seguir está contando quantos registros de adoção já foram solicitados pelo usuário para um pet:

```
$countRequestAdoption = $this->RequestAdoptions
            ->find()->where([
                'pet_id' => $petId,
                'user_id' => $userId
])->count();
```

Esta é uma regra de negócio criada no sistema para impedir que usuários façam mais de uma requisição para o mesmo animal. Caso essa quantidade seja igual a zero, o método prossegue e cria uma solicitação ou envia uma mensagem alertando o usuário de que ele já fez uma solicitação para aquele pet.

Figura 13.4: Página inicial com o botão "Eu quero".

Por enquanto, essas são as mudanças para `RequestAdoptionsController`, porém ainda temos que fechar mais uma modificação na página inicial. Será uma mudança apenas na *action* `index()` em `PetsController`. Como temos uma sessão de usuários autenticados, enquanto a sessão estiver ativa, não queremos que o usuário veja seus próprios pet cadastrados. Essa vai ser a mudança que vamos implementar no método. Vamos ao código:

```
 public function index()
    {
        $this->Flash->set('Quem ama as criaturas de Deus, ama o próprio
Deus', ['element' => 'intro']);

        $petTable = TableRegistry::getTableLocator()->get('Pets');

        if (!is_null($this->Authentication->getIdentity())) {
            $userId = $this->Authentication->getIdentity()->id;
            $query = $petTable->find()->where(function (QueryExpression
$exp, Query $q) use ($userId) {
                return $exp->notEq('user_id', $userId);
```

```
            });
            $pets = $this->paginate($query);
        }
        else{
            $pets = $this->paginate();
        }

        $this->set(compact('pets'));
    }
```

A principal mudança no código está no uso da `QueryExpression`, com a qual criamos uma `query` que vai listar todos os registros da tabela `pets` que não tenham o campo `user_id` igual ao `id` do usuário logado:

```
 $petTable->find()->where(function (QueryExpression $exp, Query $q) use
($userId) {
        return $exp->notEq('user_id', $userId);
 });
```

A `query` retornada pela pesquisa é repassada para `$this->paginate($query)` para paginar os resultados a partir dessa nova condição criada. Com essa implementação, o resultado visual pode ser observado nas imagens a seguir — a primeira mostra a tela inicial sem autenticação e a segunda, após o login:

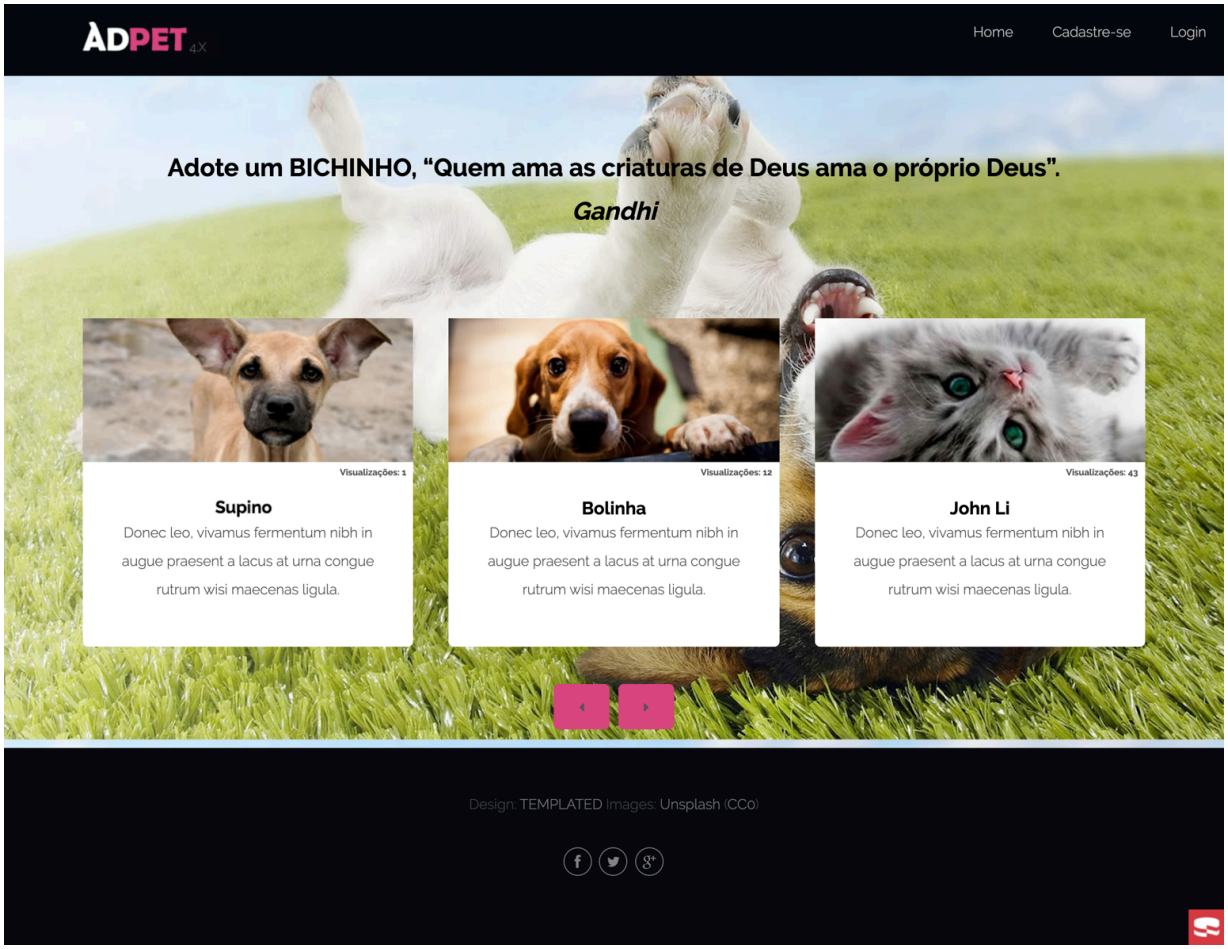


Figura 13.5: Página inicial do sistema AdPET.

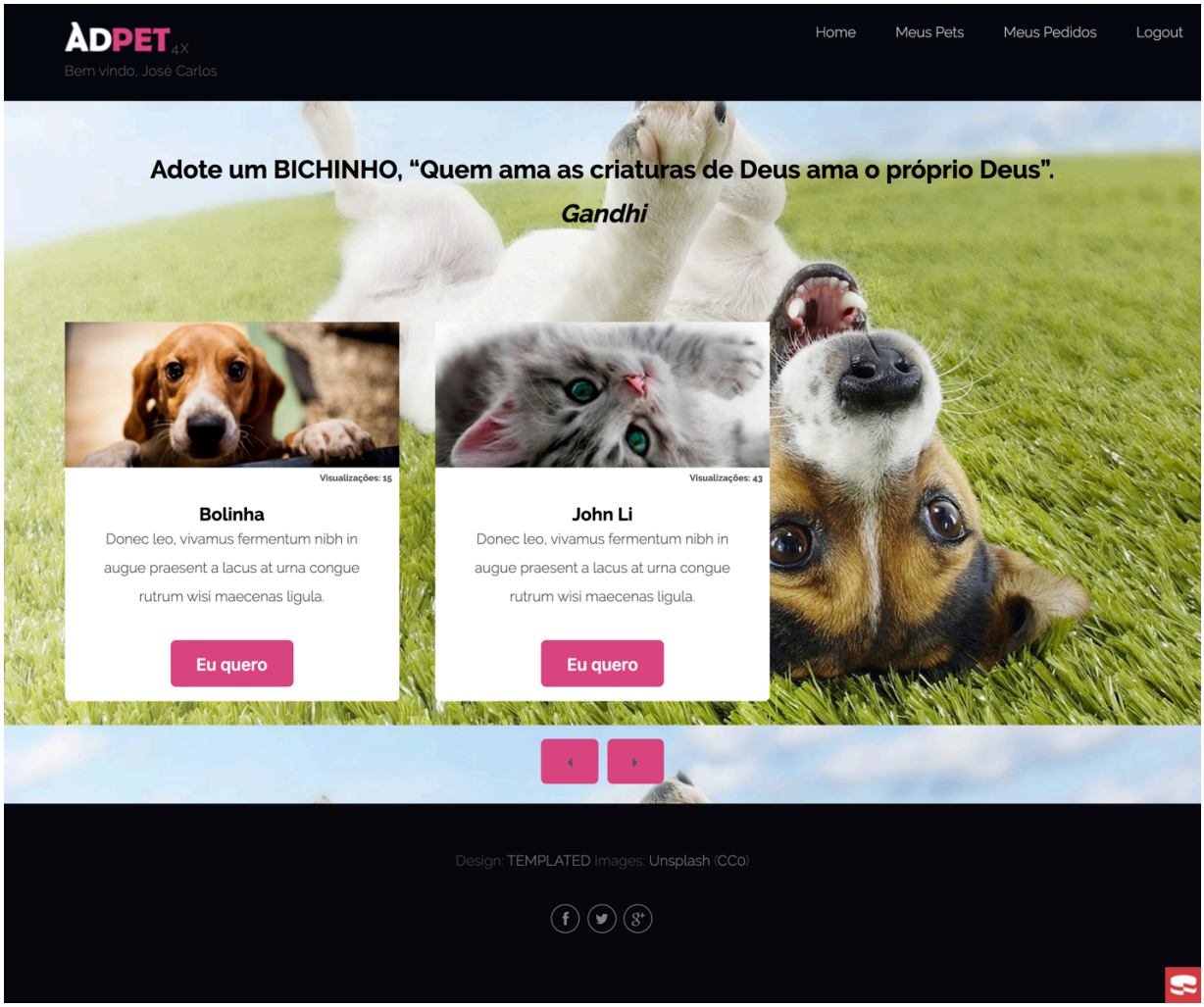


Figura 13.6: Página inicial do sistema AdPET com usuário autenticado.

Meus pedidos

Agora que disponibilizamos uma maneira para que os usuários façam pedidos de adoção, temos que criar também uma área para que os pedidos sejam gerenciados. Vamos chamá-la de `Meus Pedidos`, e ela será uma área similar a `Meus Pets`, porém, vai exibir somente os pets que receberam um pedido de adoção. O primeiro passo para iniciar o desenvolvimento dessa nova área é criar um novo item em `toolbar.php`:

```
<?php if($this->Identity->isLoggedIn()): ?>
    <li><?= $this->Html->link('Meus Pedidos', '/my-adoptions' ); ?>
```

```
    </li>
<?php endif; ?>
```

Também devemos criar uma nova linha de código em `routes.php` com o acesso que escrevemos nesse código anterior, `/my-adoptions` :

```
$builder->connect('/my-adoptions', ['controller' => 'RequestAdoptions', 'action' => 'myRequestAdoptions']);
```

Com o acesso devidamente criado, precisamos criar a *action* que vai tratar a requisição e desenvolver nela toda a lógica necessária. Em `RequestAdoptionController` , criamos o método `myRequestAdoptions()` com o código a seguir:

```
public function myRequestAdoptions(){

        $userId = $this->Authentication->getIdentity()->id;
        $adoptionsRequestTable = TableRegistry::getTableLocator()->get('RequestAdoptions');
        $results = $adoptionsRequestTable->find()->where(['user_id' => $userId])->all();

        if(count($results) > 0){
            $results = $adoptionsRequestTable->loadInto($results, ['Pets.Users', 'Users']);
        }

        $this->set('adoptions', $results);
    }
```

Na implementação do método, é feita uma pesquisa na tabela `adoptions_request` por todos os registros associados ao usuário autenticado no sistema:

```
$userId = $this->Authentication->getIdentity()->id;
$results = $adoptionsRequestTable->find()->where(['user_id' => $userId])->all();
```

A lista de resultados será enviada para *view*, porém, antes do envio, testamos se a lista tem tamanho maior do que zero em

`count($results) > 0` . Fazemos essa verificação para carregar, junto dos registros retornados pela consulta, as informações do usuário que cadastrou o pet e do usuário que solicitou a adoção:

```
$results = $adoptionsRequestTable->loadInto($results, ['Pets.Users', 'Users'])
```

O método `loadInfo()` é uma alternativa quando não usamos o `contain()` na criação da `query` , mas queremos, em algum momento, carregar as informações de algum relacionamento associado. A utilização desse método requer que a lista de resultados não seja nula ou com tamanho igual a zero. O método finaliza com o envio das informações para `templates/RequestAdoptions/my_request_adoptions.php` , cuja implementação veremos a seguir:

```
<div class="row no-collapse-1">
    <?php foreach ($adoptions as $adoption): ?>
        <section class="4u pet_block">
            <?= $this->Html->image("/img/uploads/".h($adoption->pet->profile_picture), [
                "alt" => h($adoption->user->name),
                "class" => "image featured",
                'url' => ['action' => 'view',  $adoption->id]
            ]); ?>
            <div class="views">E-mail: <?= $adoption->pet->user->email ?></div>
            <div class="box">
                <h3><strong><?= h($adoption->pet->name) ?></strong></h3>
                <p>Olá, estou no seu aguardo!!</p>
                <?php if ($this->Identity->isLoggedIn()): ?>
                <?= $this->Html->link( h('Continuar'), ['controller' => 'RequestAdoptions','action' => 'view', $adoption->id],['class' => 'button']) ?>
            </div>
        <?php endif; ?>
        </section>
    <?php endforeach; ?>
</div>
```

O leiaute da página segue o padrão que já usamos em `Meus Pets` e na página inicial, no entanto, as informações agora serão formadas a partir da tabela `request_adoptions`. Quando executada, terá a seguinte visualização:

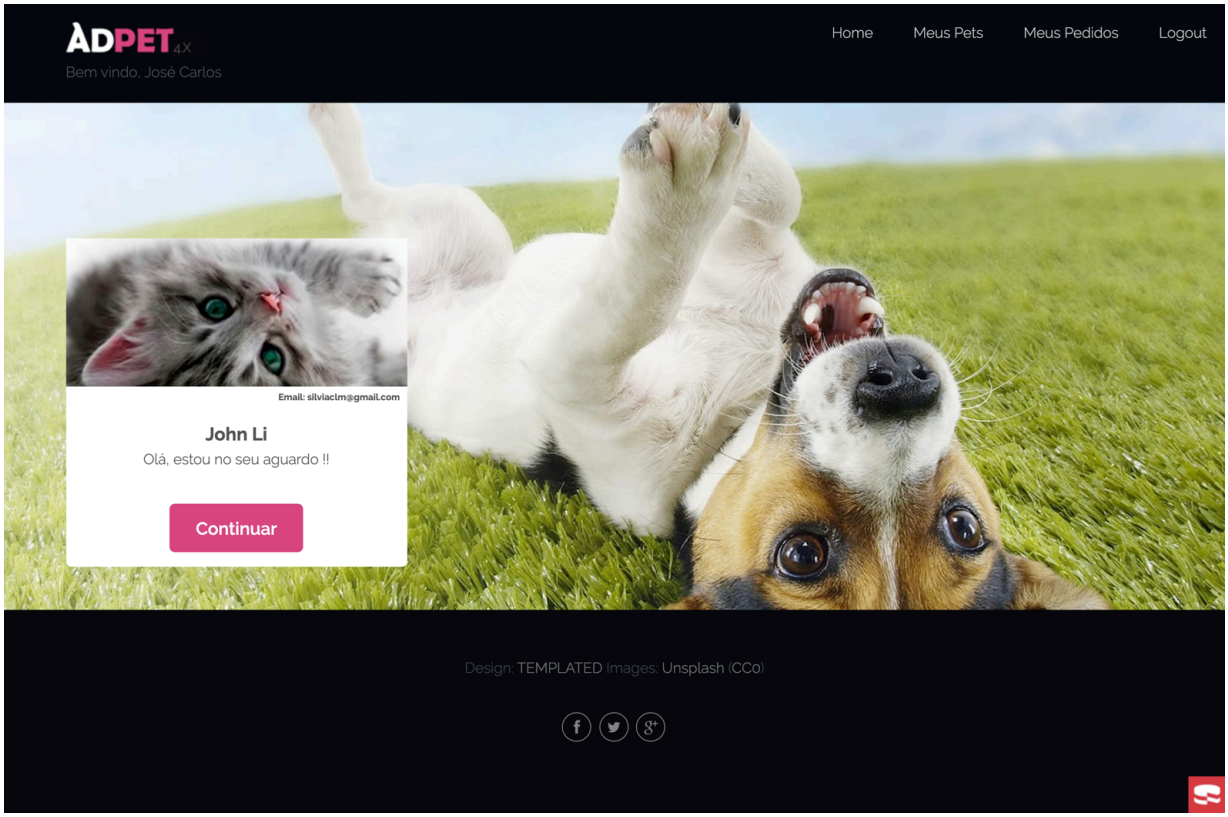


Figura 13.7: Página Meus Pets do sistema AdPET.

Negociando a adoção

No tópico anterior, criamos uma área para gerenciar os pedidos de adoção pelo usuário. Criamos um botão `Continuar`, que, quando acionado, direciona o usuário para uma página com mais detalhes do pet, do dono do animal e com um formulário que permite enviar mensagens:

```
<?= $this->Html->link( h('Continuar'), ['controller' =>
'RequestAdoptions','action' => 'view', $adoption->id],['class' =>
'button'])
?>
```

O botão direciona para a *action* `view` em `RequestAdoptions`. Nosso próximo passo será criar esse método. Pela descrição do comportamento dessa página, o método vai precisar fazer uma consulta para trazer informações das tabelas `pets` e `users`. Vejamos o código a seguir:

```
  public function view($id)
    {
         if ($this->request->is('post')) {

            $messagesTable = TableRegistry::getTableLocator()-
>get('Messages');

            $message = $messagesTable->newEntity($this->request-
>getData());
            if($messagesTable->save($message)){
                $this->redirect(['action' => 'view', 'id' => $id], 200);
            }
        }

        $adoptionsRequest = $this->RequestAdoptions->get($id, [
            'contain' => ['Pets.Users', 'Users', 'Messages.Users']
        ]);

        $this->set('adoption', $adoptionsRequest);
}
```

A primeira linha de código verifica se a requisição é um `POST`; caso verdadeiro, persiste as informações depois de criar um objeto `$message` com as informações recebidas via `POST` e, no final, redireciona o fluxo para a mesma página. O detalhe implícito aqui é que esse redirecionamento cria um novo fluxo página via `GET`, anulando os dados enviados via `POST`. O método segue montando a `query` para exibir as informações, localizando o registro de adoção pelo `id` e carregando informações das associações:

```
$adoptionsRequestTable = TableRegistry::getTableLocator()-
>get('RequestAdoptions');
```

```
        $adoptionsRequest = $adoptionsRequestTable->get($id, [
            'contain' => ['Pets.Users', 'Users', 'Messages.Users']
 ]);
```

Em `contain`, informamos para o CakePHP que precisamos também das informações das tabelas associadas. Estamos trazendo informações do usuário a partir de pet (`Pets.Users`) e do usuário que criou a mensagem (`Messages.Users`). O template para essa *action* `templates/RequestAdoptions/view.php` tem a seguinte estrutura:

```
<div id="page" class="row">
    <div id="content" class="8u skel-cell-important">
        <?php if($this->request->getAttribute('identity')->id ==
$adoption->user->id): ?>
            <section>
                <header class="major">
                    <h2><?= h($adoption->pet->user->name) ?></h2>
                    <br/>
                    <div class="img">
                        <?php if ($adoption->user->profile_picture ===
NULL): ?>
                            <?= $this->Html->image("/img/".($adoption-
>pet->user->gender === 'M' ? "man_icon.jpg" : "woman_icon.jpg"), [
                                "alt" => h($adoption->pet->user->name)
                            ]); ?>
                        <?php else: ?>
                            <?= $this->Html-
>image("/img/uploads/".h($adoption->pet->user->profile_picture), [
                                "alt" => h($adoption->pet->user->name)
                            ]); ?>
                        <?php endif; ?>
                    </div>
                    <span class="byline">
                     <strong>E-mail:</strong> <?= h($adoption->pet->user-
>email) ?> <br/>
                     <strong>Número:</strong> <?= h($adoption->pet->user-
>phone) ?> <br/>
                </header>
            </section>
```

```
        <?php else: ?>
            <section>
                <header class="major">
                    <h2><?= h($adoption->user->name) ?></h2>
                    <br/>
                    <div class="img">
                        <?php if ($adoption->user->profile_picture ===
NULL): ?>
                            <?= $this->Html->image("/img/".($adoption-
>user->gender === 'M' ? "man_icon.jpg" : "woman_icon.jpg"), [
                                "alt" => h($adoption->user->name)
                            ]); ?>
                        <?php else: ?>
                            <?= $this->Html-
>image("/img/uploads/".h($adoption->user->profile_picture), [
                                "alt" => h($adoption->user->name)
                            ]); ?>
                        <?php endif; ?>
                    </div>
                    <span class="byline">
                     <strong>E-mail:</strong> <?= h($adoption->user-
>email) ?> <br/>
                     <strong>Número:</strong>  <?= h($adoption->user-
>phone) ?> <br/>
                </header>
            </section>
        <?php endif; ?>

        <hr/>
        <section>
            <header class="major">
                <h2><?= h($adoption->pet->name) ?></h2>
                <br/>
                <div class="img">
                    <?= $this->Html->image("/img/uploads/" . h($adoption-
>pet->profile_picture),
                        ['alt' => h($adoption->pet->name)]); ?>
                </div>
                <span class="byline">
```

```
                    <strong>Amigo:</strong> <?= h($adoption->pet->name) ?
> <br/>
            </header>
            <p><?= h($adoption->pet->description) ?></p>
        </section>
    </div>
    <div id="sidebar" class="4u">
        <section>
            <div align="center" style="width: 100%">
                <header class="major">
                    <h2>Chat</h2>
                    <hr/>
                    <?php if (count($adoption->messages) == 0): ?>
                        <span>Troquem mensagens e façam um acordo que seja
o melhor para o pet</span>
                    <?php endif; ?>
                </header>
            </div>

            <?php if (!empty($adoption->messages)): ?>
                <table class="table comment-table">
                    <?php foreach ($adoption->messages as $messages): ?>
                        <tr>
                            <td width="10%"><?= $this->Html-
>image("/img/user_icon.png", ['width' => '32px', 'style'=>
'padding:4px']); ?></td>
                            <td valign="top" class="message-content">
                                <span><small><strong><?php echo $messages-
>user->name; ?> - <?php echo $messages->created; ?></strong></small>
</span><br/>
                                <?php echo $messages->content; ?>
                            </td>
                        </tr>
                    <?php endforeach; ?>
                </table>
            <?php endif; ?>
        </section>
    </div>
</div>
```

É um template maior do que já trabalhamos até o momento, mas temos mais informações para exibir. Basicamente, o código pode ser dividido em três blocos: um para exibir informações do usuário, do pet e as mensagens. Para finalizar, se você observar bem o código, não temos nenhum formulário, por isso adotamos uma estratégia diferente. O código do formulário de mensagem criamos em `templates/element/adoption_message.php` — ou seja, é um novo `Element`, o qual inserimos no template principal da aplicação. Vejamos primeiro este bloco de código para criar o formulário:

```
 <div id="footer" class="wrapper style2">
    <div class="container">
        <section style="width: 100%">
            <header class="major">
                <h2>Quase lá</h2>
                <span class="byline">Troquem mensagem e decidam o destino
do pet</span>
            </header>
            <?php echo $this->Form->create() ?>
            <?php echo $this->Form->hidden('request_adoption_id',
['default' => $adoption->id]); ?>
            <?php echo $this->Form->hidden('owner_id', ['default' =>
$this->request->getAttribute('identity')->id]); ?>
            <div class="row half">
                <div class="12u">
                    <?php echo $this->Form->textarea('content') ?>
                </div>
            </div>

            <div class="row half">
                <div class="12u">
                    <ul class="actions">
                        <li>
                            <?php echo $this->Form->button('Enviar',
['class' => 'button alt']); ?>
                        </li>
                    </ul>
                </div>
```

```
            </div>
            <?php echo $this->Form->end() ?>
        </section>
    </div>
</div>
```

É um formulário simples, cujas informações serão processadas em `view` (e já explicamos o seu funcionamento). O formulário enviará três informações: `request_adoption_id`, `owner_id` e `content` — respectivamente, o `id` do pedido de adoção no banco, o `id` do usuário que está enviando a mensagem e o conteúdo. A modificação em `templates/layout/default.php` deve ser para adicionar o seguinte trecho de código:

```
// ...
<?php if ($this->Identity->isLoggedIn()): ?>
        <?php if($this->request->params['controller'] ===
'RequestAdoptions' && $this->request->params['action'] === 'view'):?>
            <?php echo $this->element('adoption_message'); ?>
        <?php endif; ?>
    <?php endif; ?>

    <!-- Copyright -->
<?php echo $this->element('copyright'); ?>
```

Estamos apenas exibindo uma parte do código para não deixar muito extenso — o código-fonte de `templates/layout/default.php` já foi demonstrado no capítulo 10. O *element* só vai ser visível para o usuário se ele estiver autenticado e acessar a *action* `view` em `RequestAdoptions`.

Esse exemplo é interessante para você aprender como testar qual *action* estamos executando e ver que é possível colocar trechos de código que funcionam fora do template da *action* executada.

O resultado final dessa página quando acessada via botão `Continue` é:

ADPET 4X

Bem vindo, Silvia Carla

Home | Meus Pets | Meus Pedidos | Logout

DEISE CRISTINA

EMAIL: DEISE@CAKE.COM
NÚMERO:

SUPINO

AMIGO: SUPINO

Donec leo, vivamus fermentum nibh in augue praesent a lacus at urna congue rutrum wisi maecenas ligula.

CHAT

Deise Cristina - 11/20/22, 5:40 PM
Olá, que gracinha!!, ele está vacinado?

Silvia Carla - 11/20/22, 5:42 PM
Sim, ele vacinado e com muita vontade de morar com vc!!

QUASE LÁ

TROQUEM MENSAGEM E DECIDAM O DESTINO DO PET

Enviar

Design: TEMPLATED Images: Unsplash (CC0)

Figura 13.8: Página inicial do sistema AdPET com usuário autenticado.

13.3 Enviando perguntas

Para finalizar a nossa aplicação, vamos implementar uma maneira de os usuários enviarem perguntas sobre o pet antes de solicitarem uma adoção. Essas perguntas poderão ser enviadas pela página de detalhe do pet, que é a *action* `view` do *controller* `PetsController`.

Para essa funcionalidade, a *action* `view` (que, por enquanto, apenas lista as informações do pet) também vai precisar salvar mensagens — similar ao que fizemos no tópico anterior. A nova implementação vai ser:

```
public function view($id = null)
    {
        try{

            $pet = $this->Pets->get($id, ['contain' => ['Breeds','Users',
'Messages.Users'] ]);

            if ($this->request->is('post')) {

                $messagesTable = TableRegistry::getTableLocator()-
>get('Messages');

                $message = $messagesTable->newEntity($this->request-
>getData());
                if($messagesTable->save($message)) {
                    $this->Flash->success(__('Pergunta enviada com
sucesso'));
                } else {
                    $this->Flash->error(__('Não foi possível completar a
requisição'));
                }
                $this->redirect(['action' => 'view', 'id' => $id], 200);
```

```
            }
            else{
                $pet->views = $pet->views + 1;
            }

            if ($this->Pets->save($pet)) {
                $this->set('pet', $pet);
            } else {
                $this->redirect(['action' => 'index'], 404);
            }
        }
        catch(RecordNotFoundException $ex){

           $this->redirect(['action' => 'index'], 404);
        }
    }
```

Analisando esse bloco de código, vemos que o método ainda continua com a função básica, que é consultar na tabela `pets` informações do animal pelo `id`. A mudança principal é que, agora, o método pode processar informações de um `POST`, as quais serão as perguntas enviadas pelo usuário:

```
 if ($this->request->is('post')) {

                $messagesTable = TableRegistry::getTableLocator()-
>get('Messages');

                $message = $messagesTable->newEntity($this->request-
>getData());
                if($messagesTable->save($message)) {
                    $this->Flash->success(__('Pergunta enviada com
sucesso'));
                } else {
                    $this->Flash->error(__('Não foi possível completar a
requisição'));
                }
                $this->redirect(['action' => 'view', 'id' => $id], 200);
}
```

Nesse bloco de código, criamos uma instância de `MessageTable` para permitir as informações da requisição a partir de um novo objeto `$message`, que é uma `Entity`. O template `templates/Pets/view.php` também vai sofrer uma modificação, porque agora também precisa listar as mensagens que o pet recebeu. O código final do arquivo vemos a seguir:

```
 <div id="page" class="row">

    <div id="content" class="8u skel-cell-important">
        <section>
            <header class="major">
                <h2><?= h($pet->name) ?></h2>
                <br/>
                <div class="img">
                    <?= $this->Html->image("/img/uploads/" . h($pet-
>profile_picture), ['alt' => h($pet->name)]); ?>
                </div>
                <span class="byline">
                     <strong>Amigo:</strong> <?= h($pet->user->name) ?>
<br/>
                     <strong>Raça:</strong> <?= h($pet->breed->name) ?>
<br/>
                     <strong>Idade:</strong> <?= h($pet->age) ?> anos
</span>
            </header>
            <p><?= h($pet->description) ?></p>
        </section>
    </div>

    <div id="sidebar" class="4u">
        <section>
            <div align="center" style="width: 100%">
                <header class="major">
                    <h2>Perguntas</h2>
                    <br/>
                    <?php if (count($pet->messages) == 0): ?>
                        <span>Seja o primeiro a tirar suas dúvidas</span>
                    <?php endif; ?>
```

```
                </header>
            </div>

            <?php if (!empty($pet->messages)): ?>
                <table class="table comment-table">
                    <?php foreach ($pet->messages as $messages): ?>
                        <tr>
                            <td width="10%"><?= $this->Html-
>image("/img/user_icon.png", ['width' => '32px', 'style'=>
'padding:4px']); ?></td>
                            <td valign="top" class="message-content">
                                <span><small><strong><?php echo $messages-
>user->name; ?> - <?php echo $messages->created; ?></strong></small>
</span><br/>
                                <?php echo $messages->content; ?>
                            </td>
                        </tr>
                    <?php endforeach; ?>
                </table>
            <?php endif; ?>
        </section>


    </div>

    <div align="center" style="width: 100%">
        <?= $this->cell('ViewPets', [$pet]) ?>
    </div>
</div>
```

A lista de mensagens é criada no template dentro do *loop* `<?php foreach ($pet->messages as $messages): ?>` , onde `$messages` é uma variável criada a partir da consulta no *controller*.

```
this->Pets->get($id, ['contain' => ['Breeds','Users', 'Messages.Users']
]);
```

O formulário de envio também vai ser criado como um `Element` e colocado no leiaute padrão do projeto. O template do *element* tem

as linhas de código a seguir:

```
<div id="footer" class="wrapper style2">
    <div class="container">
        <section style="width: 100%">
            <header class="major">
                <h2>Pergunte ao Amigo do Pet</h2>
                <span class="byline">Tire suas dúvidas sobre o Pet</span>
            </header>
            <?php echo $this->Form->create() ?>
            <?php echo $this->Form->hidden('pet_id', ['default' => $pet-
>id]); ?>
            <?php echo $this->Form->hidden('owner_id', ['default' =>
$this->request->getAttribute('identity')->id]); ?>
            <div class="row half">
                <div class="12u">
                    <?php echo $this->Form->textarea('content') ?>
                </div>
            </div>

            <div class="row half">
                <div class="12u">
                    <ul class="actions">
                        <li>
                            <?php echo $this->Form->button('Enviar',
['class' => 'button alt']); ?>
                        </li>
                    </ul>
                </div>
            </div>
            <?php echo $this->Form->end() ?>
        </section>
    </div>
</div>
```

E, no arquivo `templates/layout/default.php` , vamos adicionar o `Element` :

```
 <?php if ($this->Identity->isLoggedIn()): ?>
        <?php if($this->request->params['controller'] === 'Pets' && $this-
```

```
>request->params['action'] === 'view'):?>
            <?php echo $this->element('question'); ?>
        <?php endif; ?>
<?php endif; ?>
```

O código final do leiaute da aplicação deve ficar assim:

```
<!DOCTYPE html>
<html>
<head>
    <?= $this->Html->charset() ?>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><?= $this->fetch('title') ?></title>
    <?= $this->Html->meta('icon') ?>
    <!--[if lte IE 8]><?= $this->Html->script('ie/html5shiv') ?><![endif]-
->
    <?= $this->Html->script(['jquery.min', 'jquery.dropotron',
'skel.min','skel-layers.min','init']) ?>
    <?= $this->Html->css(['skel', 'style', 'style-wide']) ?>
    <!--[if lte IE 8]><?= $this->Html->css('ie/v8') ?><![endif]-->
</head>

<body>
    <div class="wrapper style1">
        <?php echo $this->element('topbar'); ?>
        <?php echo $this->element('banner'); ?>
        <div id="extra">
            <div class="container">
                <?= $this->fetch('content'); ?>
            </div>
        </div>
    </div>

    <?php if ($this->Identity->isLoggedIn()): ?>
        <?php if($this->request->params['controller'] === 'Pets' && $this-
>request->params['action'] === 'view'):?>
            <?php echo $this->element('question'); ?>
        <?php endif; ?>
    <?php endif; ?>
```

```
    <?php if ($this->Identity->isLoggedIn()): ?>
        <?php if($this->request->params['controller'] ===
'RequestAdoptions' && $this->request->params['action'] === 'view'):?>
            <?php echo $this->element('adoption_message'); ?>
        <?php endif; ?>
    <?php endif; ?>

    <?php echo $this->element('copyright'); ?>
</body>
</html>
```

Então, quando clicamos em algum pet exibido, seremos redirecionados para a página de detalhe, que vai ficar com essa aparência:

ADPET 4.X

Bem vindo, José Carlos

Home | Meus Pets | Meus Pedidos | Logout

JOHN LI

AMIGO: SILVIA CARLA
RAÇA: AIREDALE TERRIER
IDADE: 1 ANOS

Donec leo, vivamus fermentum nibh in augue praesent a lacus at urna congue rutrum wisi maecenas ligula.

OUTROS BICHINHOS DA MESMA RAÇA

PERGUNTAS

Silvia Carla - 5/2/19, 2:42 PM
Olá meus amigos, que bicho é esse?

José Carlos - 5/2/19, 4:14 PM
sadasdsa dsadsadasd

PERGUNTE AO AMIGO DO PET

VAMOS, TIRE SUAS DÚVIDAS SOBRE O PET

Enviar

Design: TEMPLATED Images: Unsplash (CCo)

Figura 13.9: Página de detalhe sobre o pet.

E, assim, fechamos o nosso projeto! Obrigado!

Conclusão

Chegamos ao final do livro! Durante os capítulos, procuramos repassar todo o conhecimento de forma mais didática. Apresentamos as principais funcionalidades que, certamente, quem pretende desenvolver com o CakePHP vai precisar ter na manga.

Muito obrigado a você, leitor, leitora, que chegou até aqui, e parabéns pelo empenho. Certamente você é uma pessoa bem-sucedida, porque a busca de conhecimento faz parte da sua rotina.

Se você quiser conferir o projeto em funcionamento, ele está disponível em https://adpet.jozecarlos.com.br. O código-fonte do projeto está disponível em https://github.com/jozecarlos/adpet.

Obrigado!